

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

知名专家刘鹏教授主编畅销书《实战Hadoop》

全新升级版

实战Hadoop2.0 (第二版) 从云计算到大数据

叶晓江 刘 鹏 / 编著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>



叶晓江

云计算与大数据技术爱好者，五年Hadoop研究经验，上海互联网广告公司两年Hadoop全职开发经验。自2011年起跟随刘鹏教授从事Hadoop相关工作，并参与刘鹏教授主编的《云计算（第三版）》等书籍。现就读于南京邮电大学，攻读大数据与机器学习相关专业硕士研究生。

实战 Hadoop 2.0

从云计算到大数据

(第二版)

叶晓江 刘 鹏 编著

電子工業出版社·

Publishing House of Electronics Industry

北京 · BEIJING

内 容 简 介

本书是刘鹏教授主编的 Hadoop 编程书籍《实战 Hadoop》的第二版。Hadoop 堪称业界最经典的开源云计算和大数据平台软件。本书系统介绍了 Hadoop 2.0 生态圈的核心和扩展组件,包括:管理工具 Ambari、分布式文件系统 HDFS、分布式资源管理器 YARN、分布式并行处理 MapReduce、内存型计算框架 Spark、数据流实时处理系统 Storm、分布式锁服务 ZooKeeper、分布式数据库 HBase、数据仓库工具 Hive, 以及 Pig、Oozie、Flume、Mahout 等。

本书读者对象为各类云计算和大数据相关企业、高校和科研机构的研究人员,亦适合作为高校研究生和本科生教材。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

实战 Hadoop 2.0: 从云计算到大数据 / 叶晓江, 刘鹏编著. —2 版. —北京: 电子工业出版社, 2016.6
ISBN 978-7-121-28564-6

I. ①实… II. ①叶… ②刘… III. ①数据处理 IV. ①TP274

中国版本图书馆 CIP 数据核字 (2016) 第 073531 号

责任编辑: 董亚峰 特约编辑: 王 纲

印 刷: 涿州市京南印刷厂

装 订: 涿州市京南印刷厂

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 787×1 092 1/16 印张: 32.5 字数: 751 千字

版 次: 2016 年 6 月第 1 版

印 次: 2016 年 6 月第 1 次印刷

印 数: 4 000 册 定价: 79.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式: (010) 88254694。

第二版前言

本书第一版早在 2011 年 9 月就出版了，是国内第一本 Hadoop 编程书籍。经过 5 年发展，我们欣喜地看到，Hadoop 已经在我国遍地开花，成为云计算、大数据领域最受欢迎的开源平台。

这些年来，经过全球众多企业和个人的共同参与，Hadoop 生态圈取得了长足进步。核心版本从 1.x 升级到 2.x，并出现了以 Spark 和 Storm 为代表的全新开源软件。本书第二版的目的就是追踪最新技术，使得读者能够尽快迈进前沿。

编者从 1988 年在通信工程学院跟随谢希仁教授从事计算机网络的研究，2000 年起在清华大学跟随李三立院士从事分布式计算的研究，先后以计算机网络、网络计算、云计算和大数据为研究重点，出版了《网络计算》、《云计算》（第一、二、三版）、《实战 Hadoop》、《云计算大数据处理》、《军事信息栅格理论与技术》等书。其中，《云计算》已经成为全国高校首选教材，成为云计算从业者的“红宝书”，其免费配套 PPT 下载量逾百万次。目前，编者正联合全国多所高校和知名企业，以同样的高标准编著《大数据》教材，即将于 2016 年中出版。这些年来，编者还创办了中国云计算（www.chinacloud.cn）、中国大数据（www.thebigdata.cn）、中国物联网（www.netofthings.cn）、中国智慧城市（www.smartcitychina.cn）等网站，这些网站均在搜索引擎排名第一。希望自己所做的工作，对大家有所裨益。

下列同志参与了本书第一版的编写工作，第二版中隐含了他们的贡献。他们是：黄宜华、陈卫卫、程浩、王磊、顾荣、张贞、邓鹏、杨晓亮、郭岩岩、李浩、魏家宾、王胤然、张欣、王海坤等。本书的编写得到了云计算、大数据领域的领军企业云创大数据（网址：www.cstor.cn，微信公众号：cStor_cn，股票简称：云创数据，股票代码：835305）在软硬件环境和技术上的大力支持。在此，一并致谢！

由于编者水平有限，请读者提宝贵意见！邮箱：gloud@126.com。编者还设有微信公众号：刘鹏看未来（lpoutlook），与大家分享对科技未来的看法，并提供各种课件、资料和视频。

刘 鹏 教授
2016 年 3 月 28 日

第一版前言

1998 年, 斯坦福大学的博士生拉里·佩奇和谢尔盖·布林在车库里创建了 Google 公司。2001 年, Google 已经索引了近 30 亿个网页。2004 年, Google 发布 Gmail, 提供闻所未闻的 1GB 免费邮箱——众人还以为这是个愚人节玩笑。紧接着, Google 又发布了 Google Maps 和被称为“上帝之眼”的 Google Earth……

目前, google.com 为全世界访问量最高的站点。Google 在全球部署了约 200 多万台服务器, 每天处理数以亿计的搜索请求和用户生成的约 24PB 数据, 而且这些数据还在不断迅速增长。同时, Google 的 Android 智能手机操作系统已经拥有超过 40% 的美国智能手机用户, 而苹果仅以 8.9% 的市场份额排名第四。社交服务 Google+ 推出不到半月, 用户数量就突破 1000 万, 其增长速度罕见。数辆 Google 无人驾驶汽车已经安全行驶了至少 22.5 万公里, 没有发生过任何意外。Google 机器翻译服务能够实现 60 多种语言中任意两种语言间的互译……

是什么技术造就了这家让人惊叹的公司? 是什么样的平台在支撑这些让人匪夷所思的应用? ——全世界的人都很好奇。好在 Google 并不保守——从 2003 年开始, Google 连续几年发表论文, 揭示其核心技术, 包括 Google 文件系统 GFS、Map/Reduce 编程模式、分布式锁机制 Chubby 以及大规模分布式数据库 BigTable 等。随后, Google CEO 施密特将这类技术称之为“云计算”。所谓“云计算”, 就是用网络连接大量廉价计算节点, 通过分布式软件虚拟成一个可靠的高性能计算平台。之所以称为“云”, 是因为我们画网络图的时候, 总是将网络画成一朵云。现在, 这朵云变成了我们的“计算机”, 而我们的 PC、智能手机等则变成了它的终端, 因此称之为“云计算”。

2004 年, 正当开源搜索引擎 Nutch 和开源全文检索包 Lucene 之父 Doug Cutting 为平台的可靠性和性能深受困扰时, 看到了 Google 发表的 GFS 和 MapReduce 论文, 花了 2 年时间将之实现, 使平台的能力得到大幅提升。2006 年, Doug Cutting 加入 Yahoo!, 并将这部分工作单列形成 Hadoop 项目组。Hadoop 的名称, 并不是一个正式的英文单词, 而来源于 Doug Cutting 的小儿子对所玩的小象玩具的称呼。Hadoop 主要由以下几个子项目组成。

(1) Hadoop Common: 是支撑 Hadoop 的公共部分, 包括文件系统、远程过程调用 (RPC) 和序列化函数库等。

(2) HDFS: 提供高吞吐量的可靠分布式文件系统, 是 GFS 的开源实现。

(3) MapReduce: 大型分布式数据处理模型, 是 Google MapReduce 的开源实现。

与 Hadoop 直接相关的配套开源项目还包括如下几个方面。

(1) HBase: 支持结构化数据存储的分布式数据库, 是 BigTable 的开源实现。

(2) Hive: 提供数据摘要和查询功能的数据仓库。

(3) Pig: 是在 MapReduce 上构建的一种高级的数据流语言, 可以简化 MapReduce 任务的开发。

(4) Cassandra: 由 Facebook 支持的开源高可扩展分布式数据库。是 Amazon 底层架构 Dynamo 的全分布和 Google BigTable 的列式数据存储模型的有机结合。

(5) Chukwa: 一个用来管理大型分布式系统的数据采集系统。

(6) ZooKeeper: 用于解决分布式系统中一致性问题, 是 Chubby 的开源实现。

经过 5 年发展, 在所有的开源云计算系统里, Hadoop 稳居第一。事实上, Hadoop 是如此受欢迎, 全球已经安装了数以万计的 Hadoop 系统。不仅高校和小企业使用 Hadoop, 连 Facebook、淘宝、360 安全卫士这样的知名企业也在大规模使用 Hadoop。2007 年, Google 开始在全球推广 “Google 101” 计划, 即在全球知名高校给学生开设 Google 模式的云计算编程课程。Google 中国公司大学合作部近几年也在国内的清华大学、北京大学、南京大学、上海交大、同济大学等几家著名高校中资助开设了 MapReduce 和云计算技术课程, 本书的部分章节内容也正是在所开设课程内容的基础上形成的。有趣的是, 由于 Google 不能直接将其平台开放给学生做实验室, 于是 Google 干脆用 Hadoop 来搭建实验环境——可见 Google 对 Hadoop 的认同度。

目前国内学习和使用 Hadoop 的热情高涨。在中国云计算 (<http://www.chinacloud.cn>) 网站上做的一个调查表明, 网友将 Hadoop 作为云计算领域要学习的首选技术。然而, 目前国内 Hadoop 书籍非常匮乏, 迫切需要原著的传授 Hadoop 编程经验和解决实际问题技巧的书籍。我们的云计算技术研发团队长期战斗在存储和处理海量数据的前线, 在实践过程中积累了一些经验。为此, 我们感觉到有必要向淘宝网核心架构团队学习, 将自己积累的点滴经验贡献出来与大家分享, 于是萌生了创作此书的念头。

本书由刘鹏教授负责总体设计、组织全书写作和对内容把关调整, 黄宜华教授负责第 5 章和整体润色, 陈卫卫教授负责全书逐字审校, 程浩完成第 3、4 章, 王磊完成第 14 章, 顾荣完成第 11 章, 张贞完成第 1、2 章, 邓鹏完成第 6 章并与王胤然完成第 10 章, 杨晓亮完成第 12 章, 郭岩岩与王海坤完成第 13 章, 李浩完成第 8 章, 魏家宾完成第 7 章, 张欣完成第 9 章。南京云创存储科技有限公司对本书的编写给予了大力支持, 在紧张从事云计算系统研发的同时, 仍然派出几位研发人员参与本书写作。

衷心感谢中国电子学会云计算专家委员会、中国通信学会云计算与 SaaS 专家委员会、江苏省计算机学会、江苏省云计算论坛专家委员会和电子工业出版社对本书出版的大力支持。感谢多位知名院士、专家在百忙之中阅读书稿并给予推荐。在此拜谢我的硕士导师谢希仁教授和博士生导师李三立院士对我的辛勤培养。

本书的读者服务网页为: <http://www.chinacloud.cn/hadoop.html>, 该网页提供书中源码下载、相关信息和问题交流。如读者需要较为全面地学习云计算技术, 欢迎阅读本书的姊妹篇《云计算(第三版)》, 刘鹏主编, 电子工业出版社 2015 年 8 月出版。

由于编者水平有限, 加之时间较紧, 书中难免会存在写作不到位甚至错误之处, 敬请读者批评指正。意见和建议请发邮件到: cloudforum@163.com。

新浪微博互动交流请关注: <http://weibo.com/cloudgrid>。

解放军理工大学 刘 鹏

2011 年 9 月

目 录

第 1 章	大数据组件概述	1
1.1	Google 大数据组件	2
1.2	Apache 大数据组件	6
1.2.1	Hadoop 核心组件	7
1.2.2	基于 MR 的数据分析组件	10
1.2.3	数据库组件	16
1.2.4	BSP 组件	19
1.2.5	基于 YARN 框架组件	20
1.2.6	基于 YARN 的编程类库组件	24
1.2.7	搜索引擎组件	25
1.2.8	工作流组件	26
1.2.9	数据流组件	27
1.2.10	序列化和持久化组件	29
1.2.11	调试工具	30
1.2.12	安全性组件	30
1.2.13	兼容性组件	32
1.2.14	集群部署与管理组件	33
	习题	33
	参考文献	34
第 2 章	大数据集群	37
2.1	大数据集群简介	38
2.2	大数据集群 bigCstor	43
2.3	我的大数据集群 littleCstor	46
2.4	小结	50
	习题	50
	参考文献	51
第 3 章	集群管理工具 Ambari	53
3.1	Ambari 简介	54
3.2	使用 Ambari 部署 HDP	57
3.3	使用 Ambari 搭建 littleCstor	60

3.3.1	相关约定	60
3.3.2	制定部署规划	61
3.3.3	搭建 prelittleCstor	62
3.3.4	本地建仓	70
3.3.5	部署 AmbariServer	74
3.3.6	搭建 littleCstor	81
3.3.7	小结	104
3.4	使用 Ambari 管理 littleCstor	109
3.5	小结	110
	习题	110
	参考文献	110
第 4 章	分布式文件系统 HDFS	111
4.1	分布式存储引例	112
4.1.1	问题描述	112
4.1.2	常规解决方案	113
4.1.3	分布式解决方案	115
4.2	HDFS 简介	122
4.2.1	HDFS 逻辑架构	122
4.2.2	HDFS 物理拓扑	127
4.2.3	HDFS 部署	131
4.2.4	HDFS 其他概念	133
4.3	HDFS 接口	136
4.4	实战 HDFS Shell	139
4.4.1	HDFS 文件级命令集	139
4.4.2	HDFS 系统级命令集	141
4.5	实战 WebHDFS	148
4.5.1	WebHDFS 简介	148
4.5.2	WebHDFS 示例	150
4.6	实战 HDFS JAVA API	155
4.6.1	搭建开发环境	155
4.6.2	常规操作示例	157
4.7	实战 HDFS 大项目: 用 HDFS 存储海量视频数据	162
4.7.1	应用场景	162
4.7.2	设计实现	162
	习题	164
	参考文献	165

第 5 章 分布式资源管理器 YARN	167
5.1 分布式资源管理器引例	168
5.1.1 分布式资源管理器简介	168
5.1.2 分布式资源管理器架构	171
5.2 YARN 简介	174
5.2.1 基础概念	174
5.2.2 物理拓扑	177
5.2.3 体系架构	178
5.2.4 集群部署	187
5.3 YARN 接口	189
5.4 实战 YARN Shell	192
5.4.1 系统级命令	192
5.4.2 程序级命令	195
5.4.3 其他辅助命令	197
5.5 实战 YARN 编程	198
5.5.1 常见并行化范式	198
5.5.2 YARN 编程步骤	204
5.6 实战 YARN 编程之 DistributedShell	212
5.6.1 DistributedShell 简介	212
5.6.2 编写 DistributedShell	213
5.7 实战 YARN 编程之三大范式	220
5.7.1 DistributedShell	221
5.7.2 MapReduce	221
5.7.3 Giraph	222
习题	223
参考文献	223
第 6 章 分布式并行处理 MapReduce	225
6.1 并行化范式 M-S-R 引例	226
6.1.1 问题描述	226
6.1.2 常规解决方案	227
6.1.3 分布式解决方案	228
6.1.4 小结	234
6.2 MapReduce 简介	234
6.2.1 基本概念	235
6.2.2 编程模型	237
6.2.3 集群部署	239

6.2.4	体系架构	241
6.2.5	执行过程	245
6.3	MapReduce 接口	247
6.4	实战 MapReduce Shell	250
6.5	实战 MapReduce 编程	254
6.6	实战 MapReduce 编程之 WordCount	257
6.6.1	WordCount 代码分析	257
6.6.2	WordCount 处理过程	260
6.7	实战 MapReduce 编程之 SecondarySort	261
6.8	实战 MapReduce 编程之倒排索引	266
6.8.1	简介	266
6.8.2	分析与设计	267
6.8.3	倒排索引完整源码	269
6.9	实战 MapReduce 之性能优化	271
	习题	280
	参考文献	281
第 7 章	分布式锁服务 ZooKeeper	283
7.1	ZooKeeper 简介	284
7.1.1	ZooKeeper 应用场景	284
7.1.2	ZooKeeper 体系架构	287
7.1.3	ZooKeeper 服务模型	289
7.1.4	ZooKeeper 部署	290
7.2	ZooKeeper 接口	293
7.2.1	接口汇总	293
7.2.2	实战 ZooKeeper Shell	294
7.3	实战 ZooKeeper 编程	296
7.4	实战 ZooKeeper 之进程通信	298
7.5	实战 ZooKeeper 之进程调度系统	299
7.5.1	设计方案	299
7.5.2	设计实现	299
7.6	实战 ZooKeeper 之实现 NameNode 自动切换	305
7.6.1	设计思想	306
7.6.2	详细设计	306
7.6.3	编码	307
7.6.4	实战总结	312
	习题	312
	参考文献	313

第 8 章 分布式数据库 HBase	315
8.1 HBase 简介	316
8.1.1 体系架构	316
8.1.2 数据模型	322
8.1.3 集群部署	323
8.2 HBase 接口	328
8.3 实战 HBase Shell	329
8.4 实战 HBase API	331
8.5 实战 HBase 之综例	333
8.6 实战 HBase 之使用 MapReduce 构建索引	334
8.6.1 索引表蓝图	334
8.6.2 HBase 和 MapReduce	335
8.6.3 实现索引	336
习题	339
参考文献	339
第 9 章 内存型计算框架 Spark	341
9.1 Spark 简介	342
9.1.1 基础概念	342
9.1.2 体系架构	348
9.1.3 集群部署	359
9.1.4 计算模型	367
9.1.5 工作机制	375
9.1.6 其他特性	376
9.2 Spark 接口	377
9.3 实战 Spark Shell	380
9.3.1 集群管理	380
9.3.2 任务管理	382
9.4 实战 Spark 编程之 RDD	384
9.4.1 RDD 属性	384
9.4.2 并行化证明 RDD、调试 RDD	386
9.4.3 RDD 操作	389
9.5 实战 Spark 之 WordCount	396
9.6 实战 Spark 之 MLLib	397
习题	400
参考文献	400

第 10 章 数据流实时处理系统 Storm	401
10.1 Storm 简介	402
10.1.1 与 Hadoop 的关系	402
10.1.2 基础概念	404
10.1.3 体系架构	410
10.1.4 集群部署	414
10.1.5 计算模型	423
10.2 Storm 接口	451
10.3 实战 Storm Shell	454
10.4 实战 Storm API 之 RollingTopWords	457
习题	459
参考文献	460
第 11 章 数据仓库工具 Hive	461
11.1 Hive 简介	462
11.1.1 工作原理	462
11.1.2 体系架构	463
11.1.3 计算模型	464
11.1.4 集群部署	465
11.2 Hive 接口	469
11.2.1 接口汇总	469
11.2.2 实战 Hive Web	469
11.3 实战 Hive Shell	470
11.3.1 DDL Operations	470
11.3.2 DML Operations	471
11.3.3 SQL Operations	472
11.4 实战 Hive 之复杂语句	473
11.5 实战 Hive 之综合示例	475
11.6 实战 Hive API 接口	476
11.6.1 UDF 编程示例	476
11.6.2 UDAF 编程示例	477
习题	479
参考文献	479
第 12 章 其他常见大数据组件	481
12.1 Pig	482
12.1.1 Pig 简介	482

12.1.2 实战 Pig	485
12.2 Oozie	485
12.2.1 Oozie 简介	485
12.2.2 实战 Oozie	487
12.3 Flume	489
12.3.1 Flume 简介	489
12.3.2 实战 Flume	491
12.4 Mahout	494
12.4.1 Mahout 简介	494
12.4.2 实战 Mahout	494
习题	496
参考文献	496
附录 A 手工部署 Hadoop2.0	497
一、部署综述	498
二、部署步骤	502

“人类自有史以来的数据总量，每过 18 个月就会翻番”——“新摩尔定律”。据不完全统计，早在 2011 年，全球数据总量就已达到了 2.1ZB。IDC 更是预计 2020 年全球数据总量将超过 40ZB（40 万亿 GB），这相当于届时全球每人平均拥有 5TB 的数据量。对于如此海量的数据，传统的软件已很难处理（收集、存储、分析和应用）它们。本章所讲述的大数据组件即用来处理海量数据的一整套解决方法，不过本章只介绍这些组件，并不深入组件内部，后续章节将详细讲述部分大数据组件。

1.1 Google 大数据组件

谷歌是大数据处理技术的鼻祖，早在 2007 年，谷歌每天就要处理近 20PB 的数据量。从 2003 年至今，谷歌陆续发表多篇分布式计算论文，每一篇论文都是一场技术革新。虽然在其论文公布时，里面讲述的技术对于谷歌本身来说已经成熟到快淘汰的地步，但其依旧领先业界十几年，可以说正是谷歌在不停地引导分布式技术变革。现将论文中对应的技术以组件方式画出（图 1-1），供读者参考。



图 1-1 Google 大数据处理组件

下面从组件所处层次与功能定位角度上简单介绍各组件，由于谷歌提供的资料较少，加之对论文的理解有限，难免出错，有兴趣的读者可根据引用，深入研究谷歌论文。

1. Cluster^[1]: 集群

Cluster 指的是谷歌集群架构（Google Cluster Architecture），该论文讲述了谷歌数据中心总体体系架构，是谷歌发布的分布式方面最早的一篇论文。

2. GFS^[2]: 分布式文件系统

GFS (Google File System) 是谷歌开发的分布式文件系统。其通过在每台服务器上部署一层软件, 让集群内所有服务器存储(硬盘)空间连接到一起, 从而构成了一个统一的、无限大的文件系统。GFS 是谷歌一切上层架构的基础, 分布式数据库 BigTable、分布式批处理算法、MapReduce 等上层组件存储或处理的数据归根到底都存储在 GFS 里。

3. Chubby^[3]: 分布式锁服务

单机 OS (操作系统) 内, 不同进程间由于相互协作与竞争资源会存在同步与互斥关系。这种关系推广到集群中则是, 不同服务器上的进程会发生同步与互斥关系。显然, 单机环境下可以通过 OS 来协调这种关系, 在集群环境下, 情况则更加复杂。鉴于此, Google 通过开发了一套独立系统 (Chubby) 来协调进程间同步与互斥操作。

4. MapReduce^[4]: 分布式批处理算法

MapReduce 是一个并行化算法。在集群环境下, 即使是内网集群服务器间拷贝数据, 依旧需要大量时间(一般拷贝 1TB 数据要半小时), 可处理数据的代码却很小(一般为几 MB)。基于此谷歌提出移动代码的 MapReduce 算法。该算法通过“Map (本地处理) → Shuffle (洗牌) → Reduce (合并再处理)”三阶段, 以并行方式分布式处理存储在 HDFS 上全局数据。

5. Sawzall^[5]: 交互式 MapReduce 执行器

Sawzall 是一个提供让用户(自然人)以交互式方式操作 MapReduce 的数据分析平台。其建立在 MapReduce 基础之上, 实现了对 MapReduce 的进一步抽象。显然, 从业务上来说, 每次都写 MapReduce 太耗时耗力, Sawzall 将大量常用 MapReduce 操作(如 sum、unique 等)封装成函数形式, 并提供 MapReduce 和 GFS 接口, 这样客户端就可以直接编写脚本执行 MapReduce 和 GFS 操作了。

6. BigTable^[6]: 分布式数据库

对应于普通环境下的 RDBMS, 分布式环境下也有自己的数据库, BigTable 就是基于 GFS 实现的一个分布式数据库, 和 RDBMS 不同, 为确保底层数据的横向扩展, BigTable 是一个面向列存储的数据库。由于 BigTable 失去了太多 RDBMS 优势, 谷歌后来开发的数据库一般都是底层使用 BigTable, 中间层开发工具包提供关系型数据库处理接口。

7. Pregel^[7]: 分布式图处理框架

图是一种重要的数据结构,它能够清晰地描述多个个体之间的权重关系。谷歌内部要处理大量的图,但图的处理逻辑却很独特,不仅涉及图中边、顶点的迭代,还不停地伴随着边或顶点反馈,很明显数据之间耦合性太强,并不适合 MapReduce 这种松耦合处理机制。如何才能实现图的分布式处理呢?英国计算机科学家 Viliant 在 20 世纪 80 年代提出的 BSP (Bulk Synchronous Parallel) 并行计算模型,是当前实现对图并行处理的最优方案。谷歌基于此论文,在 Pregel 框架中实现了 BSP (整体同步并行计算模型) 处理机制,从而实现了高效分布式处理。

8. Percolator^[8]: 增量处理引擎

MapReduce 是并行处理数据的强大利器,但是它每次都是处理全局数据,在互联网上的大量信息都是在原基础上新增了一部分,比如在您更新博客时,一般都不会删除已有数据,而是在前文下追加。显然每次都用 MapReduce 进行全局处理太不合逻辑,怎样才能每次只处理新增的那部分数据呢?谷歌在 Percolator 论文中给出了一种增量处理框架。可以说 Percolator 本身就是架构在 BigTable 之上,实现增量处理的一种处理框架。

9. MegaStore^[9]: 类 RDBMS 的分布式数据库

一方面由于巨大的数据量,底层不得不采用分布式架构,可另一方面 RDBMS 在业务逻辑的表达方面又天生优于分布式数据库。可以说 MegaStore 是底层分布式(架构在 BigTable 之上),上层实现 RDBMS 接口的分布式数据库。

10. Dremel^[10]: 分布嵌套列数据库

诚然,互联网上存在着大量半结构化数据,不过若对这些半结构化数据进一步抽象,会发现大量实用数据都符合嵌套列格式,而嵌套列在分布式存储与处理领域具有天然优势,Dremel 即这样一个底层采用嵌套列结构的一个分布式数据库。

Dremel 并未像其他产品那样底层架构在 BigTable 之上,而是基于嵌套列结构,开发了一套新的存储系统。

11. Dapper^[11]: 任务跟踪机制

正常情况下,一个搜索都会转化为几个甚至几十个后台操作,Dapper 实现了查询的分解与组装。

12. Caffeine^[12]: 分布式索引架构

可以说快速搜索的核心就是索引技术, Caffeine 在数据抓取时即快速并动态建立索引。

13. Spanner^[13]: 全球统一数据库

将谷歌诸多数据中心全部接起来, 可以理解成将不同的 BigTable 数据库连起来, 构成了一个巨大的、全球统一的唯一数据库, 它的核心机制依旧是复制 (使用 Paxos 算法解决一致性问题)。

14. 组件关系

GFS 是 Google 一切上层产品的存储基础, Chubby 为集群中运行在不同服务器上的进程提供了同步与互斥机制。BigTable 则是架构在 GFS 和 Chubby 之上的一款偏底层分布式数据库, 同时 BigTable 还是其他几个高层产品 (如 Megastore、Percolator) 的底层基础。MapReduce 则是处理全局数据的强大利器, 基于它的操作集 (函数集) 加上其他实用类库即构成产品 Sawzall。

由于图处理的独特机制, Google 在 Pregel 中采用了 BSP 处理机制来处理图结构。同样, 由于互联网上大量信息都是增量的, Google 开发了 Percolator 来快速处理新增的数据。

数据库方面, Dremel 是一款在数据存储结构方面 (使用嵌套列) 有别于 BigTable 的新颖数据库。Megastore 则是架构在 BigTable 之上, 操作上尽量 RDBMS 化的分布式数据库。

Caffeine 并不特指某一个产品, 它是诸多产品的组合, 实际上它几乎包含了从爬虫、建立索引到 PageRank 的所有过程。不过, 在理解时可以把它简化为建立索引的过程, Caffeine 放弃了使用 MapReduce 来构建多层静态索引, 转而使用 Percolator 增量机制来实现动态实时单层索引。

Spanner 则更加雄心勃勃, 它把 Google 所有数据中心全部连到一起成为一个巨大全局数据库, 理解时可以认为 Spanner 全局 master 管理所有 BigTable 数据中心。

需要指出的是, MapReduce 处理机制和数据库、BSP 处理模型、增量处理机制并不矛盾, 需要对全局数据进行离线处理时, MapReduce 依旧是最好的处理工具。

在分布式处理方面谷歌贡献卓越, 针对谷歌的每一篇论文, 许多公司或组织都会开发相应的开源产品。下节将介绍的 Apache 软件基金会就是当今世界最著名的开源组织。

1.2 Apache 大数据组件

Apache^[14]软件基金会 (Apache Software Foundation) 是当前全球最有名的开源组织, 其最具盛名的就是 Web 服务器软件 apache (与组织名相同, 但一个是产品, 一个是组织), 此外, 它还针对一些有名的商业软件开发了相应的开源版本, 如微软 Office 的开源版 openOffice 等。

2000 年, Apache 开启了一个和搜索相关的 Lucene^[15]项目, 此项目主要功能是提供全文文本搜索函数库。2002 年 Apache 在 Lucene 基础上新开发了网络爬虫 (Crawler) 和 Web 模块, 并称新项目为 Nutch^[16]。但在 Nutch 开发过程中 Apache 遇到了无法将计算任务分配到多机执行的问题, 此间 (2004 年前后) 恰巧谷歌发布论文 GFS 和 MapReduce, 参考此论文, 于是有了 Nutch 版 NDFS (Nutch Distributed FileSystem) 和 MapReduce。由于 Nutch 是一个包括爬虫、索引和查询等功能的完整搜索系统, 而 NDFS 和 MapReduce 模块只是让 Nutch 任务可以以分布式方式在多机上执行, 这两个模块更像一个分布式基础架构, 故在 2006 年 Apache 将 NDFS 和 MapReduce 抽出, 成立一个新的项目, 称为 Hadoop^[17]。

随着技术和实践的发展, Hadoop 本身也在发展并完善着, 工业界称 Hadoop1.X 及其以前的版本 (0.23.X 除外) 为 Hadoop1.0, 称 Hadoop2.X 及其以后版本为 Hadoop2.0, 两个版本在 Hadoop 架构上有着巨大改变, 本书主要讲解当前主流版本 Hadoop2.0。

Hadoop (Hadoop2.0, 本书中的 Hadoop 皆指 Hadoop2.0) 本身只包括分布式存储 HDFS、分布式资源管理框架 YARN 和分布式全局数据处理引擎 MapReduce。从功能上看, Hadoop 不可能做完所有事情, 它其实是一个分布式基础架构, 提供分布式环境下最基本最核心功能——分布式存储和分布式集群资源管理。显然我们还可以在 Hadoop 外层开发大量实用组件, 让 Hadoop 支持更多功能。此时, Hadoop 及其生态圈组件即构成大数据处理平台, 这也是本书中将主要讲解的部分。

近年来, 围绕 Hadoop 的外围组件发展迅速, 比如就如何将普通环境下的数据导入 HDFS 就出现了 Flume、Sqoop、Chukwa、Kafka 等组件。下面以功能为中心, 对各个组件进行简单介绍。Apache 组织下和 Hadoop 相关的组件如图 1-2 所示, 其中白色框为成熟顶级组件, 灰色框为孵化组件。一般情况下, 孵化组件的相关文档配备齐全后, 即会转成成熟组件。请注意, Hadoop 只包含 HDFS, YARN 和 MapReduce, 图中单独画出的实体框如 HBase 和 ZooKeeper 都是独立的项目。

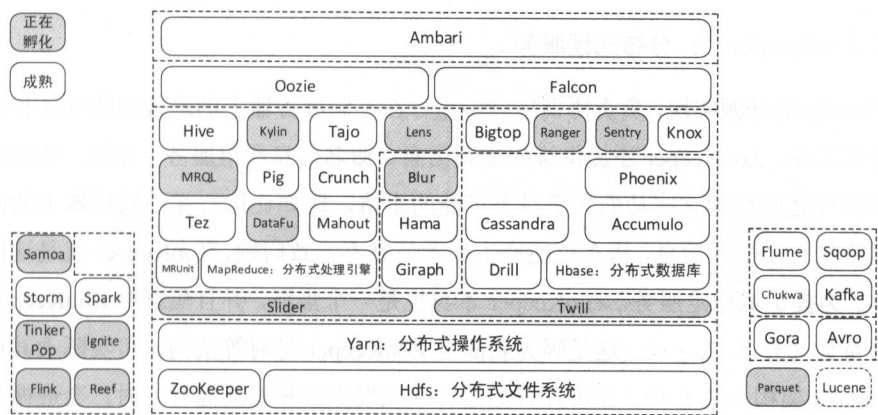


图 1-2 Apache 大数据组件层次关系逻辑示意图

1.2.1 Hadoop 核心组件

正如前文所示，Hadoop 本身只包含 HDFS、YARN 和 MapReduce，不过，由于 HBase 和 ZooKeeper 功能非常重要，一般也将这两个组件分为 Hadoop 核心组件。

1. HDFS^[18]：分布式文件系统

GFS 的开源实现。HDFS（Hadoop Distribute FileSystem）采用 master/slave 架构，主服务器运行 master 进程 Namenode，从服务器运行 slave 进程 DataNode。它将集群中所有服务器存储空间连接到一起，构成了一个统一的、海量的存储空间。图 1-3 是 HDFS 的物理拓扑图，本书第 4 章将深入讲解 HDFS。

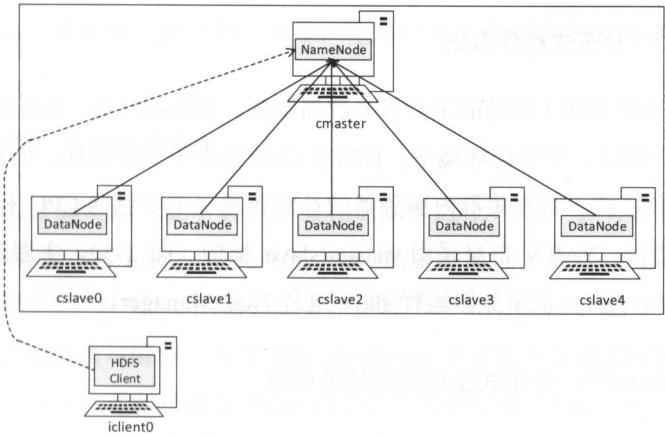


图 1-3 HDFS 物理拓扑图

2. ZooKeeper^[19]: 分布式锁服务

Chubby 的开源实现，负责协调集群中运行在不同服务器上的进程间的互斥和同步操作，除此之外，ZooKeeper 还提供维护配置信息、命名信息和组服务。显然，让集群中的分布式程序之间协调这些功能几乎是不可能的事情，比如让运行在不同机器上的四个进程共同维护一个配置信息，极有可能产生配置信息不一致问题，ZooKeeper 就是这样一个功能体。为提供最稳定服务，ZooKeeper 本身也是一个集群，并且集群中 ZooKeeper 个数只能为奇数（3, 5, 7, …）。这是因为表面上 ZooKeeper 是对等架构，可实际上其依旧是 master/slave 架构（图 1-4），集群在开始运行（或发生故障重新运行）时，会遵循少数服从多数原则，选出一个领导。ZooKeeper 内部采用 Paxos 算法来实现数据一致性，可以把它看成 Paxos 算法的代码实现。作为分布式环境下最重要的组件之一，本书将在第 7 章深入讲解 ZooKeeper。

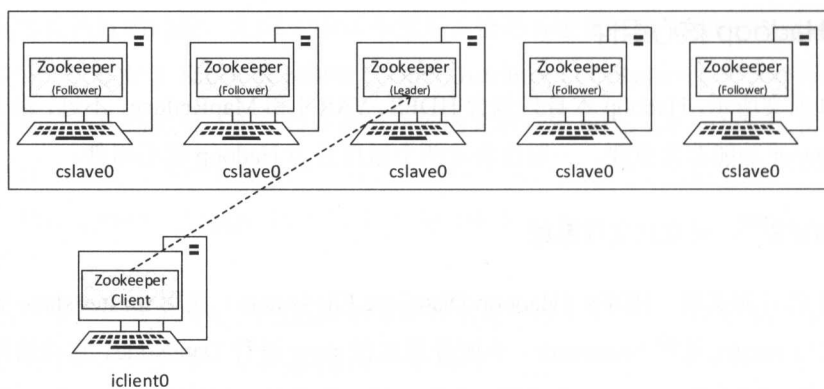


图 1-4 ZooKeeper 物理拓扑图

3. YARN^[20]: 分布式操作系统

普通 OS（操作系统）的功能主要为处理器管理、存储器管理、设备管理、文件管理、作业管理和用户接口。在集群环境下，HDFS 已经负责了文件管理，而设备概念较弱，故 YARN 主要负责统一管理集群内服务器的计算资源（主要包括 CPU 和内存资源）、作业调度和用户接口。YARN 自身采用 master/slave 架构（图 1-5），主服务器运行 master 进程 ResourceManager，从服务器运行 slave 进程 NodeManager。

4. MapReduce^[21]: 分布式全局数据处理引擎

谷歌 MapReduce 的开源实现，通过“Map（本地处理）→Shuffle（洗牌）→Reduce（规约处理）”三阶段（图 1-6），并行处理存储在 HDFS 上的全局数据。

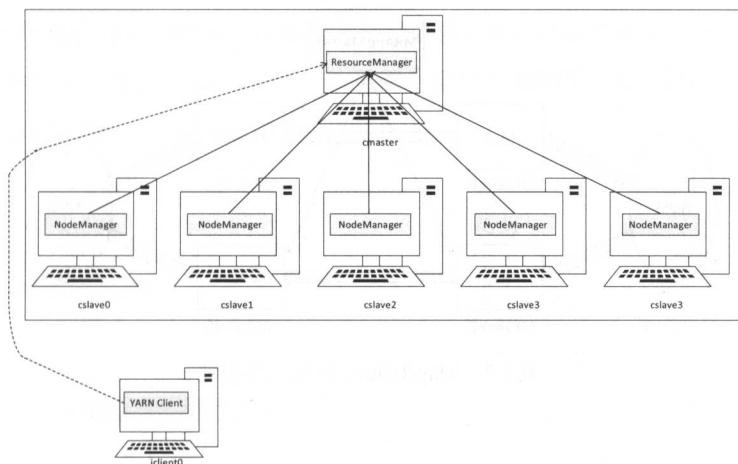


图 1-5 YARN 物理拓扑图

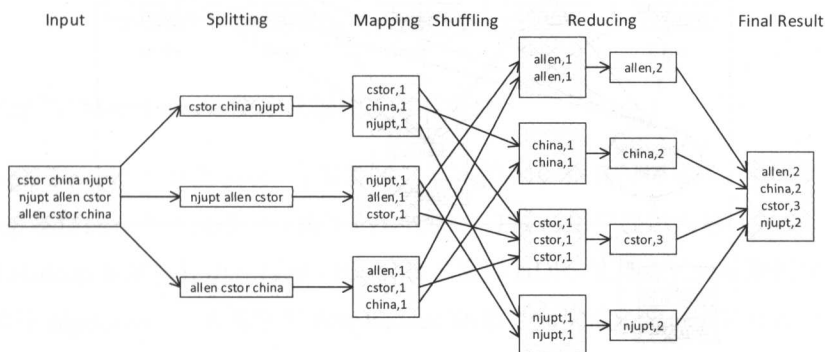


图 1-6 MapReduce 三阶段

显然要实现上述三阶段，从代码层次来说，依旧需要一个主进程（Master）和诸多从进程（Worker）来实现上述过程。其中，主进程负责指挥整个过程，从进程负责执行 Map 和 Reduce（图 1-7）。

5. HBase^[22]：分布式数据库

BigTable 的开源实现，底层数据存储存储在 HDFS 之上，上一层采用 master/slave 架构（图 1-8），主服务器运行 master 进程 HMaster，从服务器运行 slave 进程 HRegionServer。Hmaster 主要管理用户对 Table 的增删改查，管理所有 HRegionServer，它不会参与 Table 中具体数据的查看修改等操作；每个 HRegionServer 负责具体存储索引信息、缓存数据，修改日志信息，并且它是客户读写 Table 信息的具体执行者；Table 中存储的实实在在信息，一般都被 HRegionServer 下放到 HDFS 中存储。

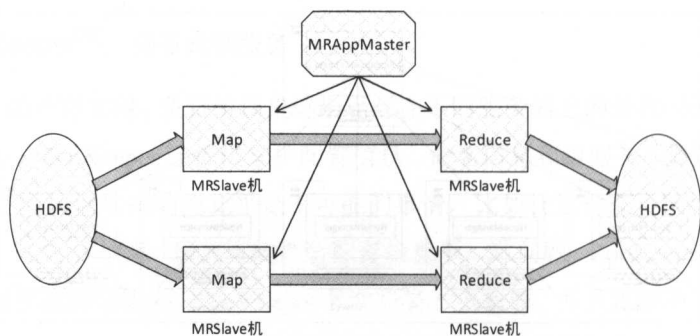


图 1-7 MapReduce 框架三大模块

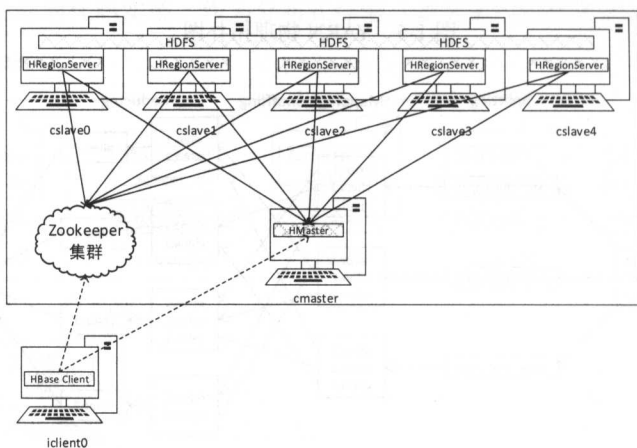


图 1-8 HBase 物理拓扑图

1.2.2 基于 MR 的数据分析组件

虽然说 MapReduce 代码的开发量不大，但和 SQL 语句比起来，编写 MapReduce 代码依旧不够灵活。鉴于此，一些公司或组织将他们开发的 MapReduce 函数集（如 add 函数，unique 函数）添加到一起，再加上一个翻译器和 HDFS 工具集，即构成了基于 MapReduce 封装的组件（如 Hive、Mahout 等）。随着技术的发展，基于 MapReduce 的组件越来越多，下面介绍基于 MapReduce 的常用组件。

1. Tez^[23]：DAG 作业的计算框架

假如一个作业需要 4 个有前后依赖关系的 MR（MapReduce）完成，比如执行顺序只能为“Job1→Job2→Job3→Job4”，则正常编写时就要写 4 个 MR，且系统在执行时，Job1、

Job2、Job3 结果都会存入 HDFS，这显然完全没必要；Tez 可以将这 4 个 Job 配置成一个 DAG 作业（图 1-9），且前 $(n-1)$ 个作业结果都不会存入 HDFS，大大提高了作业执行效率。此外，也可在 Pig，Hive 中使用 Tez（图 1-9）。

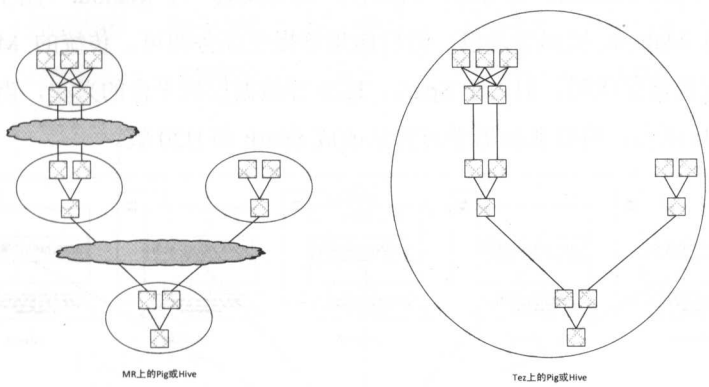


图 1-9 DAG 型 MapReduce 任务

2. Pig^[24]：MapReduce 脚本化操作工具类组件

Google Sawzall 的开源实现，主要提供交互式的 MR 和 HDFS 操作。实际上 Pig 相当于 Hadoop 集群的一个智能终端（图 1-10），对外，Pig 向用户提供大量内置函数，对内，Pig 调用 Hadoop 集群执行用户程序。Pig 封装了常用 HDFS 实用类（如将数据存入 HDFS 的存储操作 `pigstorage`）、大量常用 MapReduce 函数（如均值函数 `AVG({(int)})`）并提供了一个运行 Pig 的执行环境，方便实用，不需要每次都编写 MR 代码，对编程能力要求较低。

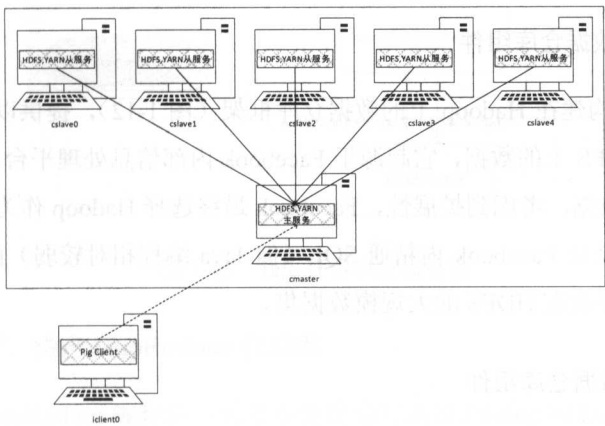


图 1-10 Pig 物理拓扑示意图

3. Mahout^[25]: MapReduce 版机器学习算法代码库

简单地说，将大量经典算法（如 Naive Bayes、k-Means）组合到一起，加上一些执行辅助工具即构成 Mahout。使用时，只在客户端上安装一个 Mahout（图 1-11），用户代码里直接使用 Mahout 机器学习包，然后向集群提交任务即可。传统的 Mahout 只包含 MapReduce 版机器学习代码，但由于 Spark、H2O 等数据挖掘平台的兴起，当前 Mahout 里正在使用 Scala 语言，将经典机器学习算法改成 Spark 和 H2O 版。

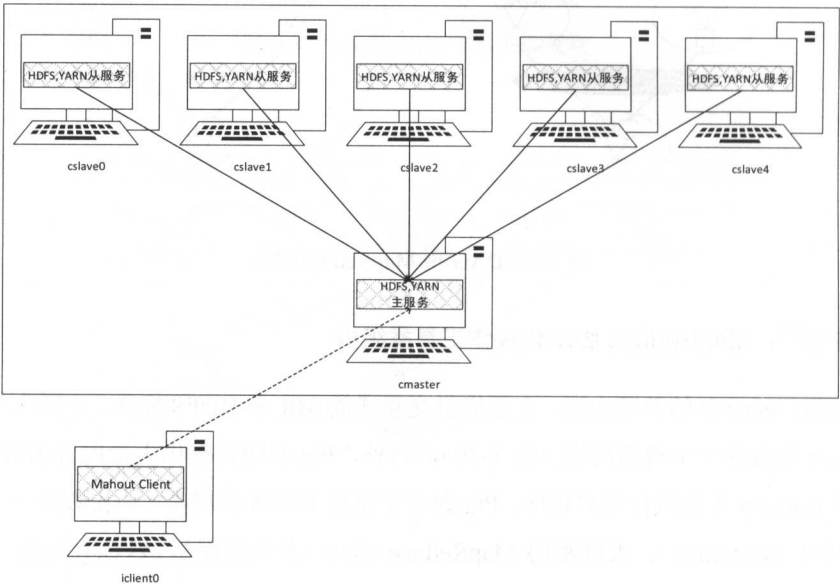


图 1-11 Mahout 物理拓扑示意图

4. Hive^[26]: 数据仓库组件

Hive 是一个构建在 Hadoop 上的数据仓库框架（图 1-12），提供以类 SQL 方式管理和查询存储在 HDFS 上的数据，它起源于 Facebook 内部信息处理平台。由于需要处理大量新兴社会网络数据，考虑到扩展性，Facebook 最终选择 Hadoop 作为存储和处理平台。Hive 的设计目的是让 Facebook 内精通 SQL（但 Java 编程相对较弱）的分析师能够以类 SQL 的方式查询存放在 HDFS 的大规模数据集。

5. Tajo^[27]: 数据仓库组件

Tajo 是一个由韩国大学数据库实验室基于 YARN 开发的开源分布式数据仓库。其设计思想类似于 Tenzing，由于充分借鉴了 MapReduce 和 DataBase 的优势，其具有 Hive 的扩展性和容错性好的优点，同时性能也比 Hive 高许多。它具有低延迟、高可伸缩的特点，

并且还提供专用查询和 ETL 工具。Tajo 采用 master/slave 架构，master 机运行主进程 Tajomaster，slave 机运行 slave 进程 TajoWorker（图 1-13）。

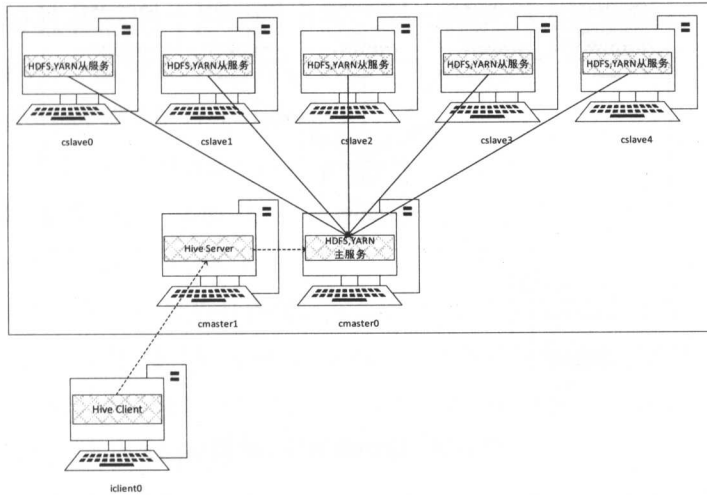


图 1-12 Hive 物理拓扑示意图

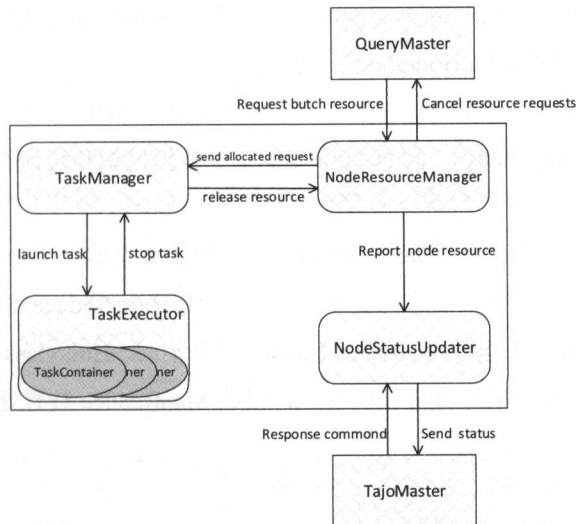


图 1-13 Tajo 任务执行流程图

6. Crunch^[28]: 特定 MapReduce 代码库

直接使用 MapReduce 编程的一大难点便是编写诸如 Joining 和 Data Aggregation 这类操作，Crunch 相当于 Hadoop 集群的一个智能终端（图 1-14），其提供了大量数据聚合类函数（如 Join, union）。它的 APIs 特别适合处理诸如“时间序列”、“序列化对象格式”（如使用 Protocol 或 Avro 对原始数据进行序列化后数据）、HBase 的行和列族这类非关系

型数据。

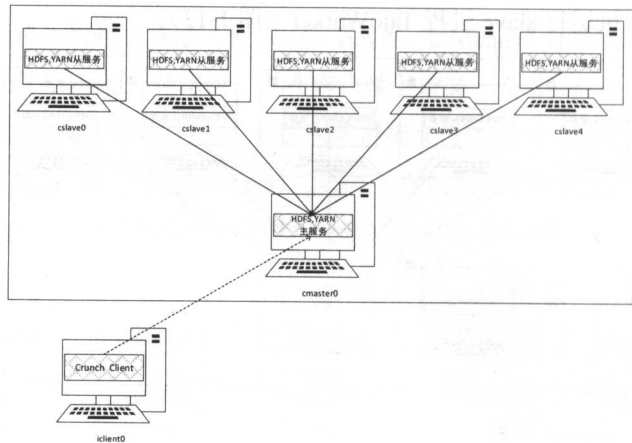


图 1-14 Crunch 物理拓扑图

7. DataFu^[29]: 强大的 MR 类库

DataFu 是由 LinkedIn 在 2011 年开发的一个基于 MapReduce 的类库 (图 1-15), 包含 “DataFu Pig” 和 “DataFu Hourglass” 两部分。

开发初期, DataFu 很类似 Pig 中的用户定义函数集 (UDF), 不同的是 Pig 中一般都是通用的 UDF (如 Piggybank), 而 DataFu 则侧重于数据挖掘和统计类的函数 (如分位数计算和取样方法等)

2013 年 10 月, 一个名为 “DataFu Hourglass” 的新类库加入了 DataFu。Hourglass 是一个增量处理框架, 不过需要注意的是 Hourglass 使用 MapReduce 来处理增量数据, 这点和基于 BSP 实现的几个组件 (Giraph、Hama) 完全不同。Hourglass 在 HDFS 中保存上一个作业完整状态, 并用它来处理新的输入, 以让 MapReduce 具有处理增量数据能力。

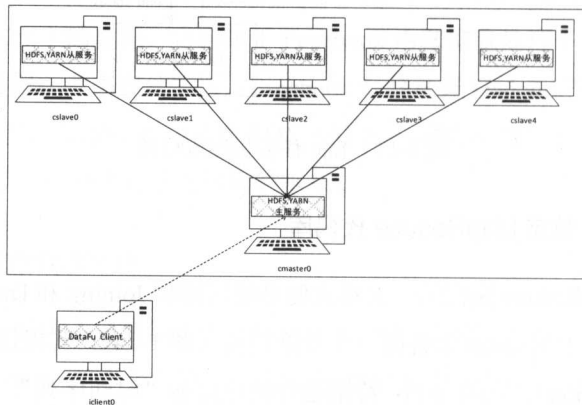


图 1-15 DtaFu 物理拓扑图

8. MRQL^[30]: 在 Hadoop、Hama、Spark、Flink 之上的类 SQL 查询系统

MRQL (MapReduce Query Language) 是一个查询处理和优化系统, 适用于大规模分布式的数据分析。MRQL 语言风格类 SQL, 基于 YARN, 构建在 Hadoop、Hama、Spark、Flink 之上, 能以以下四种模式来运行。

- 以 MapReduce 模式运行在 Hadoop 上;
- 以 BSP 模式运行在 Hama 上;
- 以 RDD 方式运行在 Spark 上;
- 以 Flink 模式运行在 Flink 上。

MRQL 语言灵活强大, 能胜任各类日常数据分析任务, 它不仅善于分析存储在 HDFS 里的二进制数据, 还能直接分析 XML、JSON、CSV 等各种原生场景数据。和 Hive、Pig 相比, MRQL 提供了更加强大的查询功能, 能够以多种模式操作更加复杂的数据类型, 而 Hive、Pig 实质上都以 MapReduce 方式操作数据。

得益于 MRQL 灵活的语言, 用户可以只用 MRQL 语句就能写出像 PageRank、K-means 聚类、矩阵分解 (Matrix Factorization) 这类复杂的数据分析任务。而 MRQL 后台自动将写好的查询语句编译成高效的代码, 并以合适的模式运行。

9. Kylin^[31]: 多维数据的 OLAP 分析引擎

Kylin (麒麟) 是一个分布式 OLAP (联机分析处理) 多维数据分析引擎, 其由 eBay 的韩卿 (中国人) 领导开发。Kylin 构建在 Hadoop 上, 提供类 SQL 分析查询功能, 用户可以以 SQL 方式在线或离线处理数据流 (图 1-16)。不过 Kylin 最大的优势是, 其可用于执行多维数据的 OLAP 分析, 且在使用过程中, Kylin 自动为用户屏蔽了多维立方 (OLAP Cube), 极大地简化了产品使用。

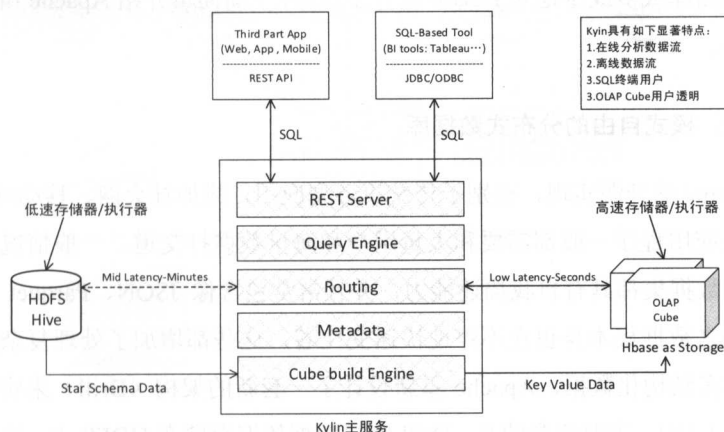


图 1-16 Kylin 体系架构

10. Lens^[32]: 统一的分析接口

Lens 底层架构在 Hadoop 相关组件和传统数据仓库之上，为前端用户提供了一个统一的分析平台（图 1-17）。它不仅能够使用 MR 和 Hive 分析 HDFS 里面数据，还能直接使用类 SQL 查询数据立方里的数据。

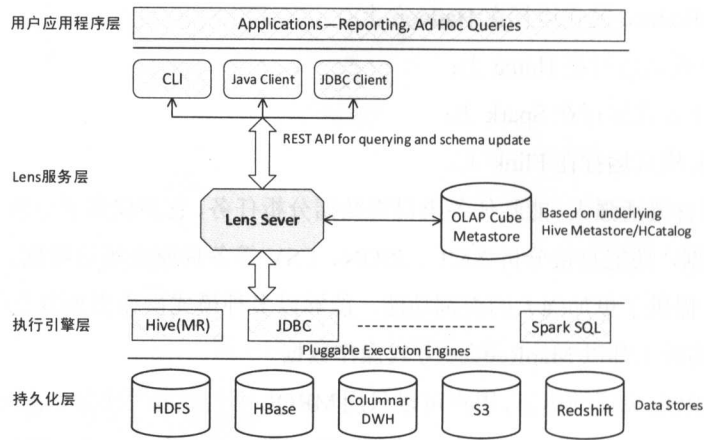


图 1-17 Lens 体系架构

1.2.3 数据库组件

若想快速实现对数据读取操作几乎都必然要数据库技术支持，谷歌于 2006 年发表论文 BigTable，但 BigTable 的开源实现并不是只有 HBase。诚然，HBase 是当前最为成熟、应用最广的分布式数据库，但其采用<Row,ClomunFamily>数据模式不能解决一切问题，其他分布式数据库或多或少地对 HBase 进行了改进，下面简单介绍 Apache 组织下和数据库相关的组件。

1. Drill^[33]: 模式自由的分布式数据库

谷歌 Dremel 的开源实现，有别于传统事务型应用，诸如社会网、移动网、Web、物联网等现代的应用程序一般都需要和大量用户和海量数据打交道。一般情况下，这些应用程序对应的数据集都具有自我描述能力，并且常常包含像 JSON、Parquet 之类的复杂数据结构，加之数据集本身也在不停地快速变化着，这些都增加了处理复杂性。基于这种快速变化的多结构化数据，Apache 重新设计了一套新的架构（Drill）来实现低时延原生生态查询（图 1-18）。需要注意的是，Drill 底层数据依旧存储在 HDFS 上，这让它具有了强大的水平扩展性。

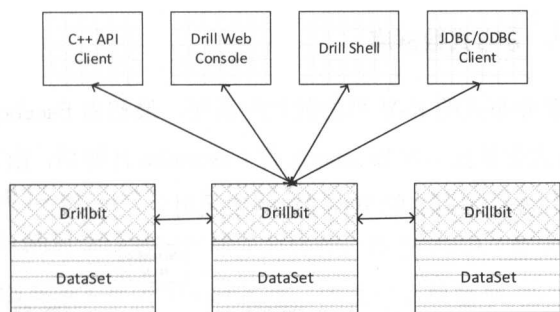


图 1-18 Drill 体系架构

2. Accumulo^[34]: 安全性更强的分布式数据库

NSA(美国国家安全局)于 2008 年基于 BigTable 开启了分布式数据库项目 Accumulo，开发它的部分原因是为了解决大数据安全问题，2011 时 NSA 将它捐赠给了 Apache。Accumulo 的底层架构在 Hadoop(HDFS、YARN 和 MR)、ZooKeeper 和 Thrift 之上(图 1-19)，它在数据安全上做了严格的限制。事实上，HBase 并不支持在其最小逻辑单元 Cell（行键、列、时间戳唯一确定）上设置访问控制权限（最小设置单元为 Cell 的上一层列族层），这大大降低了表中数据的安全性。而 Accumulo 能够提供 Cell 级别的细粒度访问控制属性。实现时 Accumulo 扩展了 BigTable 的数据模型，添加了一个元素，以提供单元格级别的、强制的基于属性的访问控制（ABAC）。在导入数据时可以使用“可见性控制”来标记所有导入 Accumulo 中的数据。而当我们读取数据时，由于访问控制策略中的“可见性控制”，我们将仅能看到我们有权看到的内容。

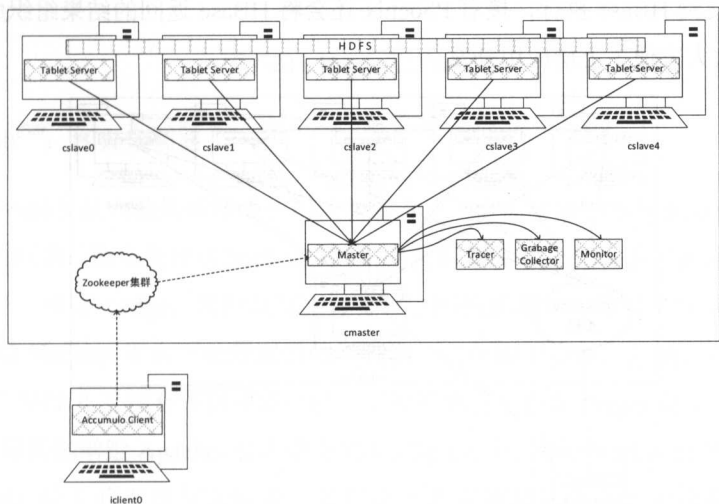


图 1-19 Accumulo 物理拓扑图

3. Cassandra^[35]: 分布式数据库

Cassandra 是一套分布式的 K-V 型的数据库系统，最初由 Facebook 开发，用于存储邮箱等比较简单的格式化数据。在 Facebook 将 Cassandra 开源后，由于 Cassandra 良好的可扩展性（图 1-20），被许多著名的 Web 2.0 网站采用。

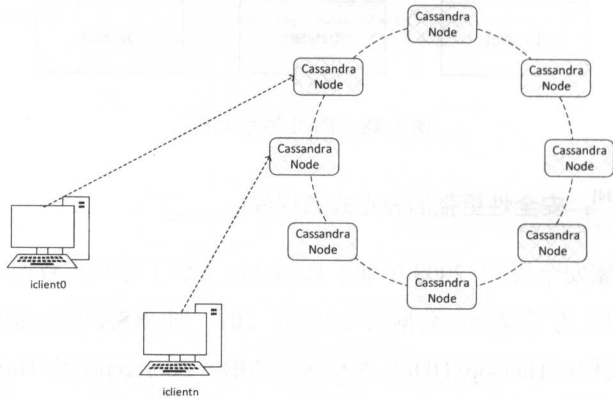


图 1-20 Cassandra 物理拓扑图

4. Phoenix^[36]: NoSql 版的数据库连接驱动包 (JDBC)

Phoenix 是架构在 HBase 之上的关系数据库层（图 1-21）。Java 代码可使用 JDBC 连接 MySQL 数据库，Phoenix 就相当于 HBase 版的 JDBC，使用它可以高效地访问 HBase 数据库。当我们使用 Phoenix 读写 HBase 时，它会把我们的 SQL 查询编译成一系列 HBase Scans，并提交到 HBase 执行，接着 Phoenix 还会将 HBase 返回的结果组织成标准 JDBC 结果集，这大大方便了上层应用开发。

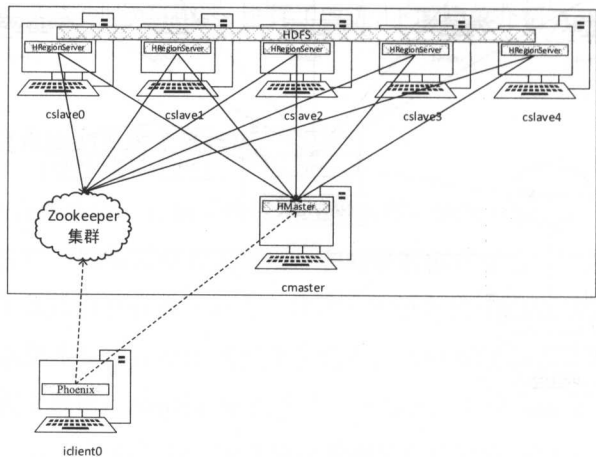


图 1-21 Phoenix 物理拓扑

1.2.4 BSP 组件

并行模型 MapReduce 和分布式数据库 HBase 并不是万能的，比如 MapReduce 并不适合做含有大量迭代的科学计算与机器学习任务，但在 BSP（整体同步并行计算模型）计算模型下，却能够高效实现诸如矩阵运算、图迭代、机器学习，这种大规模科学计算与迭代任务。BSP 模型由哈佛大学 L. Valiant 教授于 1990 年提出，并经牛津大学 W. F. McColl 教授进一步发展。BSP 为并行计算提供一个像串行计算的冯·诺依曼模型同样稳定的模型，从而实现并行计算的可扩展性、可移植性和可预测性。

1. Hama^[37]：通用 BSP 计算引擎

Hama 一个是架构在 Hadoop 之上的通用 BSP 计算引擎（图 1-22）。类似于大多数分布式计算架构，Hama 也采用 master/slave 架构，主服务器运行 master 进程 BSPmaster，从服务器运行 slave 进程 GroomServer。Hama 的底层数据一般都存储在 HDFS 之上，在执行 BSP 任务时还使用 ZooKeeper 管理 BSPPeer 的同步，以实现 Barrier Synchronisation 机制。

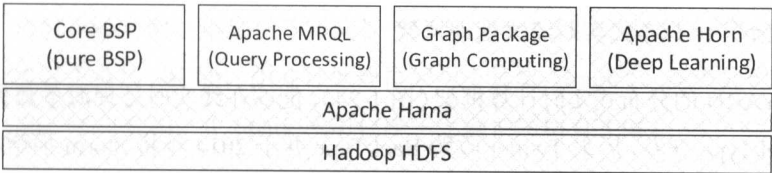


图 1-22 Hama 层次结构图

2. Giraph^[38]：图处理利器

图形（Graph）是大数据领域最热门的关键词之一。为了能够更好地分析人群、位置和事件之间的联系，图形处理引擎和图形数据库利用系统节点（Node，例如 Facebook 用户的兴趣爱好）和边（edge，用户及其兴趣爱好之间的联系）对数据进行分析。

Giraph 是 Hadoop 平台下处理图的强大利器，它起源于雅虎。起初，雅虎使用 MPI 来处理图，但 MPI 几乎没有任何容错功能，于是雅虎借鉴谷歌 Pregel 论文开发了 Giraph 并于 2013 年将其捐赠给 Apache。让人意外的是，Giraph 开源后，Facebook（而不是 Yahoo）却成了 Giraph 最大的支持与贡献者。我想这主要应该是这两个公司的业务不同吧，Facebook 在开发图谱搜索（Graph Search）服务时，看中了 Giraph 的速度和可伸缩性，选用它作为后台处理框架（图 1-23），来处理数万亿次用户及其行为之间的连接。

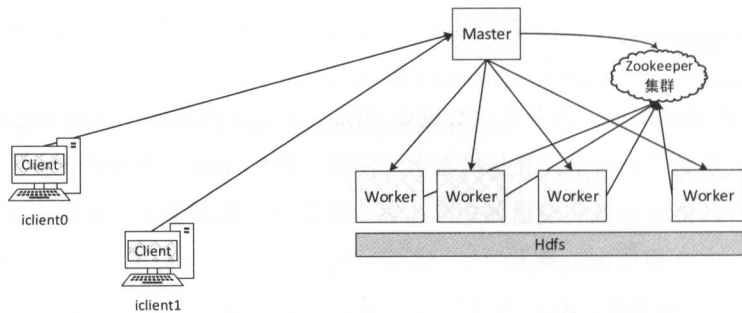


图 1-23 Giraph 体系架构

1.2.5 基于 YARN 框架组件

作为一个批处理框架，MapReduce 以及基于 MapReduce 的组件（如 Hive 等）适合离线处理全量数据，如果需要在线实时处理数据或者高效执行机器学习算法，可以选用一些新出现的框架，本节即讲述这些新出现的基于 YARN 的机器学习或实时处理框架。

1. Storm^[39]：分布式实时流式处理框架

基于 YARN 的分布式实时计算框架(图 1-24)，能够在线实时处理海量数据流。Storm 最初由 BackType 开发，后来 Twitter 收购 BackType 并于 2014 年将 Storm 捐赠给 Apache。Storm 采用 master/slave 架构，主服务器运行 master 进程 Nimbus，从服务器运行 slave 进程 Supervisor。Nimbus 与 Supervisor 之间由 ZooKeeper 协调。因为要实时地处理流式数据，除非手工停止，否则 Storm 任务会一直不停地执行下去。Storm 为 Hadoop 带来了可靠的实时数据处理能力。

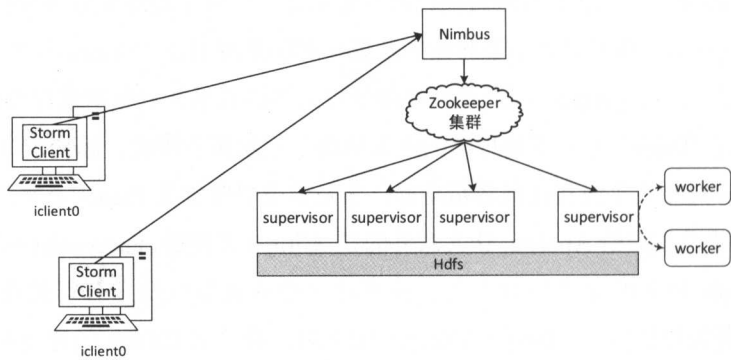


图 1-24 Storm 体系架构

2. Samoa^[40]: Storm 的算法包

如果说 Mahout 是 MapReduce 的机器学习算法包，那么 Samoa 就是 Storm 的“Mahout”。目前 Samoa 支持本地、Storm、S4 和 Samza 这几个平台。开发与测试代码时，建议使用本地模式，实际生产时则使用 Storm 平台，这样可以保持生态圈完整性。目前，Samoa 主要支持如下算法：

- 垂直 Hoeffding 树分类器；
- 自适应 Model Rules 回归器；
- Bagging and Boosting；
- 分布式流聚类；
- 分布式流频繁项挖掘。

3. Spark^[41]: 内存型 Hadoop

Spark 是用 Scala 语言开发的一套分布式内存型计算框架（图 1-25），其起源于 2009 年加州伯克利分校 AMP 实验室的一个研究性项目，于 2010 年开放源码，并在 2013 年捐赠给 Apache。虽然它自身用 Scala，但其支持包括 Java 在内的几乎所有主流语言。Spark 围绕着其核心 RDD（Resilient Distributed Dataset，弹性分布式数据集）开发了一系列分布式 API，可以直接对数据集进行分布式处理。在 Spark 之上还有 SQL 查询工具 Spark SQL，机器学习工具集 MLlib，图并行处理框架与工具集 GraphX，流处理框架 Spark Streaming，用户可以在同一个 Spark 应用程序中无缝使用所有类库。

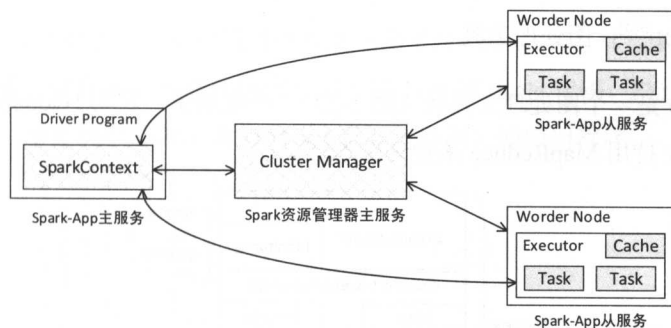


图 1-25 Spark 体系架构

4. Reef^[42]: 基于 YARN 的可保留计算执行框架

Reef（Retainable Evaluator Execution Framework）是微软基于 YARN 开发的一款可保留计算执行框架。事实上，在 YARN 上编写应用程序难度很高，Reef 紧贴 YARN 层，用来统一底层，并提供大量实用类库来简化更高层应用开发难度（图 1-26）。

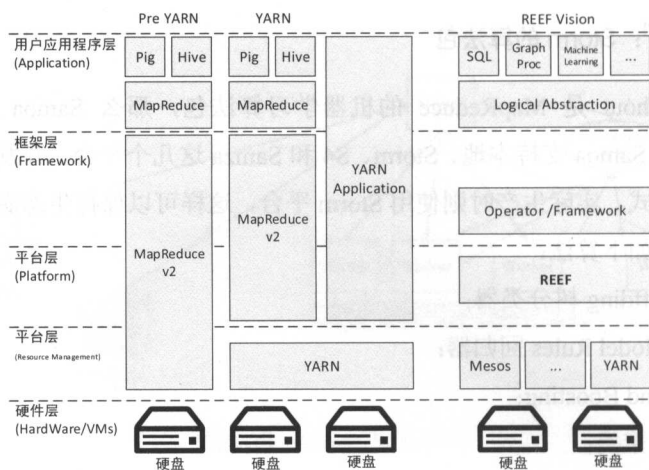


图 1-26 Reef 层次结构

和 Apache YARN、Apache Mesos、Google Omega、Facebook Corona 相比，Reef 提供了一个集中式管理层抽象来管理底层的分散到各处的数据，这样大大方便更高层系统开发。此外，Reef 在设计之初就考虑到了图计算、机器学习应用程序，显然这类应用一般都需要在已经分配资源的机器上长期保存数据，以供多次使用。

一般来说，运行在 YARN 上的应用程序都需要多个数据处理过程，比如正常应用都会经历数据洗牌 (Shuffle)、组通信 (Group Communication)、数据聚合 (Aggregation)、校验 (Checkpointing) 等过程。显然，为每个上层应用都设计这么一套规则太不切实际，Reef 内部实现了大量数据处理类库，这样上层应用就可以直接调用类库了。

5. TinkerPop^[43]：图处理工具

ThinkerPop 是一个图处理工具包 (图 1-27)，严格地说，它与 Hadoop 关系不大，不过 ThinkerPop 支持用 MapReduce 来处理图迭代。

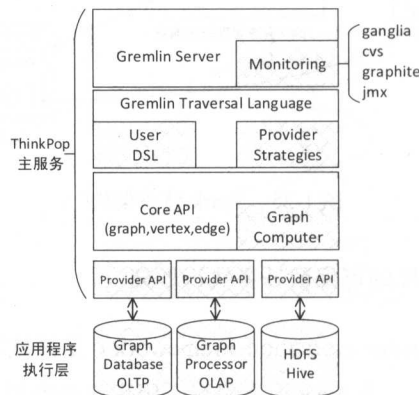


图 1-27 ThinkerPop 体系架构

6. Ignite^[44]: 内存数据组织框架

Ignite 内存数组组织框架是一个高性能、集成的分布式内存型计算和数据交换平台，其常用于大规模的数据集的实时在线处理。Ignite 为不同应用和不同的数据源之间提供一个高性能、分布式内存中数据组织管理的框架（图 1-28）。其起初由 GridGain 开发，该公司核心业务即是提供分布式内存片处理技术。由于当前主流 Web 服务底层依旧架构在 RDBMS 之上，Ignite 将 NoSQL 和 RDBMS 结合，并提供统一内存数据接口。鉴于内存网络的应用前景，Ignite 应用会越来越广阔。

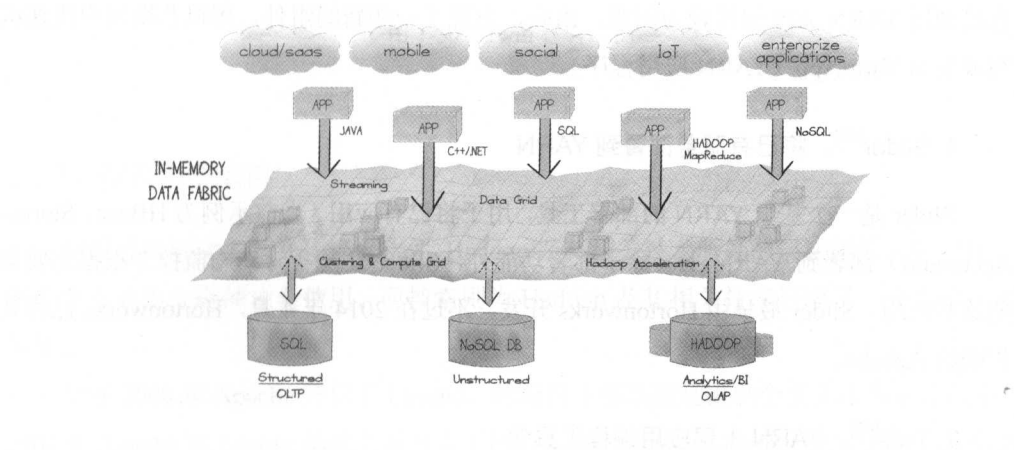


图 1-28 Ignite 体系架构（取自官网）

7. Flink^[45]: 基于 YARN 的快速可靠的大规模数据处理引擎

Flink 是一个高性能、分布式的通用数据处理平台（图 1-29），其提供了 Java 和 Scala 编程接口并内置了优化器，自动优化用户编写的 Java 或 Scala 代码。Flink 主要用来处理包含大量迭代、增量迭代或者大量 DAGs（有向无环图）的应用程序。

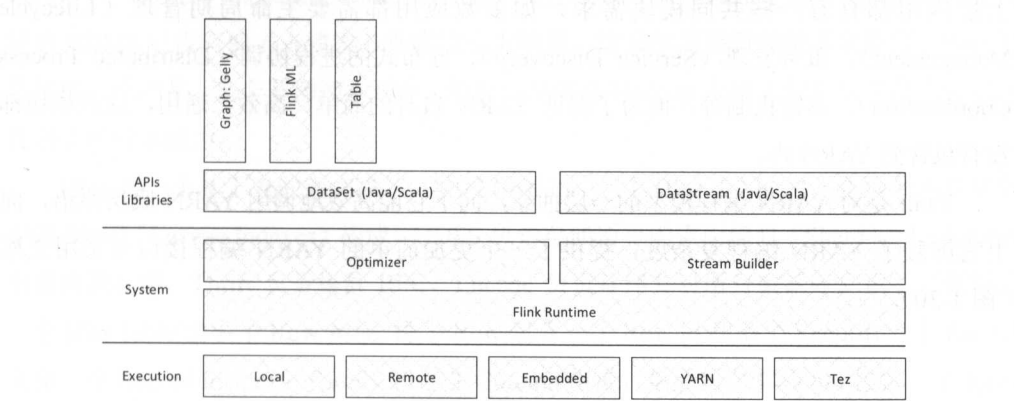


图 1-29 Flink 体系架构

利用内存数据流, Flink 可快速处理大量数据。此外, Flink 内置自定义的一套内存管理组件、序列化框架和类型引用引擎, 这能确保它在集群内存耗尽时依旧保持高效的性能。Flink 基于 YARN, 不仅可以读取 HDFS 数据, 还能读取 HBase 和 Hive 里的数据。

1.2.6 基于 YARN 的编程类库组件

为确保 YARN 精简高效, YARN 自身代码量、复杂度有严格限制, 这就导致了用户直接调用 YARN API 编程较为困难。由此, 出现了一些新的组件, 用以帮助用户快速编写或发布 Yarn-App (YARN 应用程序)。

1. Slider^[46]: 将已有服务部署到 YARN

Slider 是一个基于 YARN 的部署工具, 用于将已有应用 (当前示例为 HBase、Storm、Accumulo) 部署到 YARN 上。除了部署功能外, Slider 还能实时动态监控并根据需要调整这些应用。Slider 最早由 Hortonworks 开发, 不过在 2014 年 4 月, Hortonworks 已将其捐赠给 Apache。

2. Twill^[47]: YARN 上层应用编程工具包

Twill 是一个用来简化高层分布式应用开发的工具包, 它能让开发者更多地关注他们的应用逻辑, 而不是陷入 YARN 的底层通信接口。

YARN 是 Hadoop 平台的通用资源管理框架, 在它之上可以运行各类应用。可是虽然 YARN 很强大, YARN 的学习门槛却非常高, 由于它太偏底层 (大量 RPC 调用), 即使我们花费大量时间精力去深入学习它, 也不一定能高效驾驭它。事实上, 基于 YARN 的上层应用都有着一些共同模块需求, 如多数应用都需要生命周期管理 (Lifecycle Management)、服务发现 (Service Discovery)、分布式的进程协调 (Distributed Process Coordination)、容错机制等, 但为了保证 YARN 自身的简单、高效、通用, 这些模块都没有包含到 YARN 内。

Twill 是对 YARN 编程模型的一层抽象, 向下它能高效地调用 YARN 底层原语, 向上它屏蔽了 YARN 编程复杂度, 提供了一个更加简单的 YARN 编程接口与实用类库 (图 1-30)。

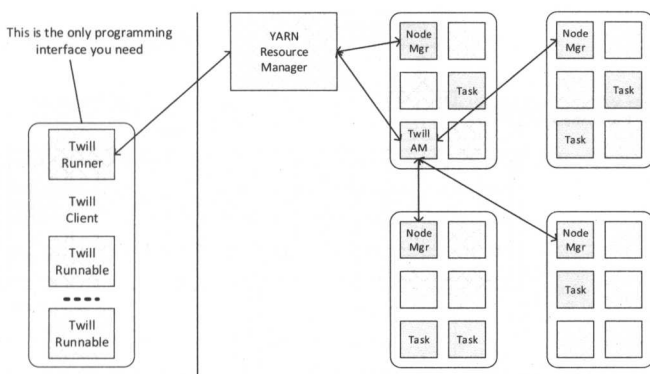


图 1-30 Twill 任务示例

1.2.7 搜索引擎组件

如果按照层次来划分计算机软件，Hadoop 及其相关技术所处的层次很低，即使 Hive，都需要专业的分析师才会使用，而搜索则为 Hadoop 及其相关技术开辟了一个新的应用领域。

早在 2000 年 Apache 开启了 Lucene，此项目主要功能是提供全文文本搜索函数库。2002 年 Apache 在 Lucene 基础上新开发了网络爬虫（Crawler）和 Web 相关模块，并称为 Nutch。在 Nutch 开发过程中又遇到了无法将计算任务分配到多机执行问题，此间恰巧谷歌发布论文 GFS 和 MapReduce，于是有了 Nutch 版 DFS 和 MapReduce。由于 Nutch 是一个完整的系统，它包括爬虫、索引、查询等，而 DFS 和 MapReduce 只是让 Nutch 任务可以以分布式的方式在多机上执行，故将 DFS 和 MapReduce 抽出，并称为 Hadoop。

Blur^[48]项目开启于 2011 年，它基于 Lucene、Hadoop、Thrift、ZooKeeper，它和 Hadoop 来自同一“血统”，可以说是 Nutch 的重构。现在很多组织机构拥有大量的数据，可是传统的 RDMS 已没有能力去建索引并搜索这么大数据，这必然需要新的技术来处理。Blur 就是这样一个产品，是个开源搜索平台，用来给存储在 Hadoop 上的大规模数据建立索引并提供实时搜索能力。

Blur 是一个基于表的查询系统，物理存储上，一个 Blur Table 由一系列分布在集群中的 Shard（Lucene Indexes）构成，HDFS 负责存储所有的索引信息（Shard），ZooKeeper 负责协调集群，Thrift 负责集群 RPC，Lucene 负责存储并对所有数据建立索引；逻辑上一个 Blur Table 由多个 Row 构成，每个 Row 包含一个 Row Id 和多个 Record，一个 Record 又由一个 Record Id、一个 Family 和许多 Column 组成，最后每个 Column 包含一个 Key、Value 对（图 1-31）。

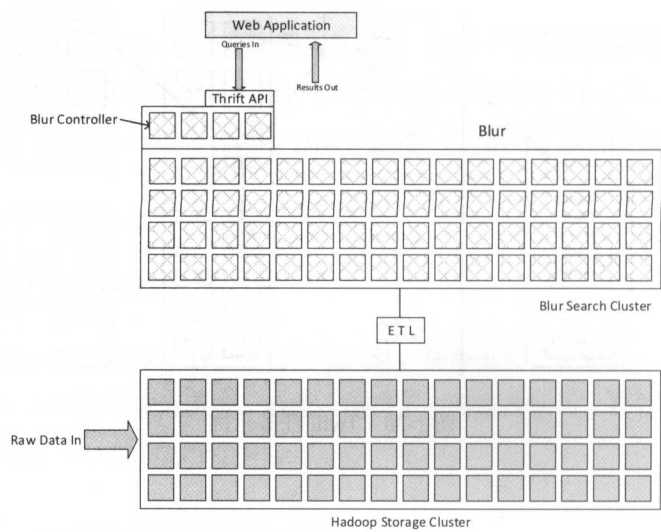


图 1-31 Blur 体系架构

1.2.8 工作流组件

公司的一个数据分析任务处理流程是“PythonApp→MRApp→HiveApp→JavaApp→WekaApp→ShellApp”，这就是所谓的工作流。显然，一个项目不可能只有一个 MapReduce 程序，工作流场景很常见。

1. Oozie^[49]：工作流组件

现实业务中的计算任务一般都不止一个 MR，比如现有一个计算任务，该任务处理流是“M1→R1→Java1→Pig1→Hive1→M2→R2→Java2→Mahout1→Java3”，当编写完各个模块后，用户可能会写 python 脚本将各个模块串起来一起执行。Oozie 即提供了这样一套模板，让用户可以以 xml 格式，定义处理流（图 1-32）。Oozie 的能力还不止这些，它还提供了时间和数据促发器，让用户可以用它来执行定时任务、数据促发任务。

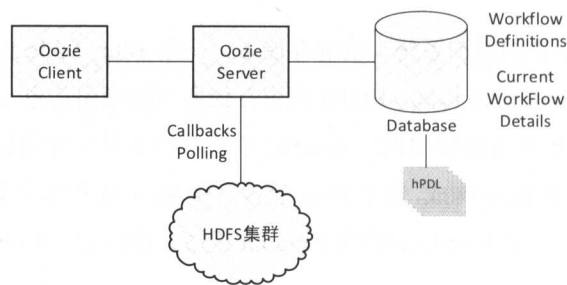


图 1-32 Oozie 任务示例

2. Falcon^[50]: 数据管理框架

Falcon 是一种数据管理框架（图 1-33），用于简化数据生命周期管理并处理 Hadoop 上的管道（实际上底层借助了 Oozie）。它可让用户编排数据移动、管道处理、灾难恢复（借助 HCatalog）以及数据保留 workflows。

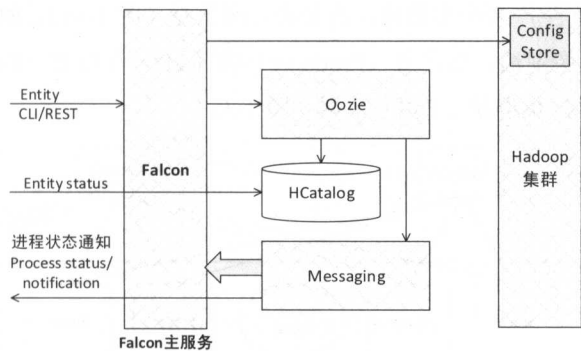


图 1-33 Falcon 层次位置图

1.2.9 数据流组件

创建好大规模 Hadoop 集群后，集群中的数据从何而来？是否可以将 MySQL 里的数据导入 HDFS，又如何才能够将线上服务器产生的日志实时导入 HDFS？下面将介绍的组件就是用来将数据导入导出 HDFS/Hive/HBase。

1. Flume^[51]: 数据像水一样流往各处

Flume 是一个分布式高性能高可靠的数据传输工具，它可以以简单的方式将不同数据源的数据导入某个或多个数据中心，典型应用是将众多生产机器日志数据实时导入 HDFS（图 1-34）。除了数据传输功能外，Flume 更像一个智能的路由器，内部提供了强大的分用、复用，断网续存功能。Flume 使用特别简单，如果将数据看成水，那 Flume 就是沟渠，使用时，我们只指定 Flume 的源头和终点即可。

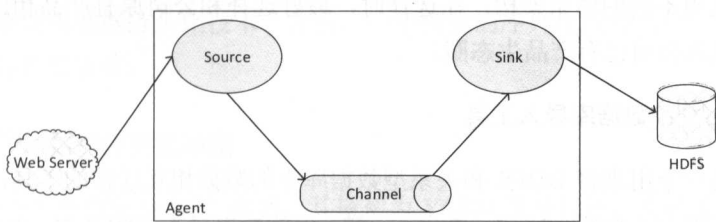


图 1-34 Flume 体系架构

2. Kafka^[52]: 一个基于发布-订阅机制的高性能、分布式消息系统

Kafka 源于职业社交网站 LinkedIn, 该项目于 2012 年 12 月开源, 现已成为 Apache 顶级项目。对于一个职业社交平台来说, 有些用户订阅了他们的邮件, 有些没有, 有些用户订阅了工程类邮件, 有些可能订阅了财经类邮件。Kafka 就是这样一个采用发布-订阅机制的消息系统, 生产者产生数据, 消费者订阅数据 (图 1-35)。如果说 Flume 像沟渠的话, 那 Kafka 更像湖泊, 生产者 (Producer) 向 Kafka 集群推 (Push) 数据, 订阅者 (Consumer) 从 Kafka 集群拉 (Pull) 数据 (图 1-35)。

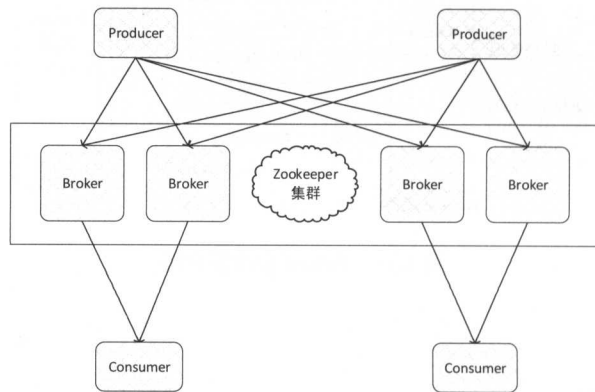


图 1-35 Kafka 体系架构

3. Chukwa^[53]: 分布式数据收集系统

Chukwa 是一个分布式数据收集系统, 其由雅虎开发并捐赠给 Apache。在现实经济活动中, 即使一个中等规模的网站, 每天都会产生数量庞大的日志文件。诚然, MapReduce 很适合处理如此庞大的数据, 但问题是, 这些数据怎么就到了 HDFS? Chukwa 即可高效完成此事, 它可以实时监控在线服务器 (生产进程) 产生的日志信息, 并实时将日志导入 HDFS/Hive/HBase (图 1-36)。在导入前, 用户可定制一套符合业务逻辑的数据流, 这样在导入时就能按此逻辑对数据进行去重、排序、筛选, 从而大大方便上层应用。

显然, Flume、Kafka、Chukwa 这三个组件功能上很相似, 应用场景也有很多相同之处。不过, 它们架构设计差异很大, Flume 采用对等架构, Kafka 和 Chukwa 采用 master/slave 架构。由于采用不同的体系架构, 在选择时, 最好选择和公司原有产品相似的产品, 这样能够无缝融入公司已有产品生态圈。

4. Sqoop^[54]: 数据库导入工具

Sqoop 是一个用来将 HDFS 和关系型数据库中的数据相互迁移的工具, 其可以将一个关系型数据库 (MySQL、Oracle、Postgres 等) 中的数据导入 HDFS 中, 也可以将 HDFS 的数据导入关系型数据库中 (图 1-37)。

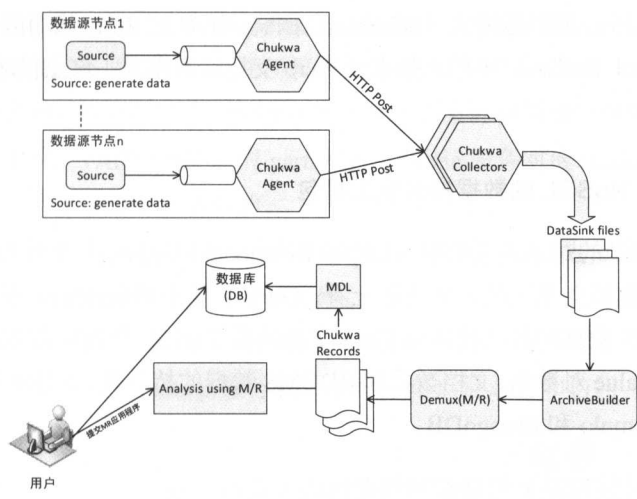


图 1-36 Chukwa 任务示例

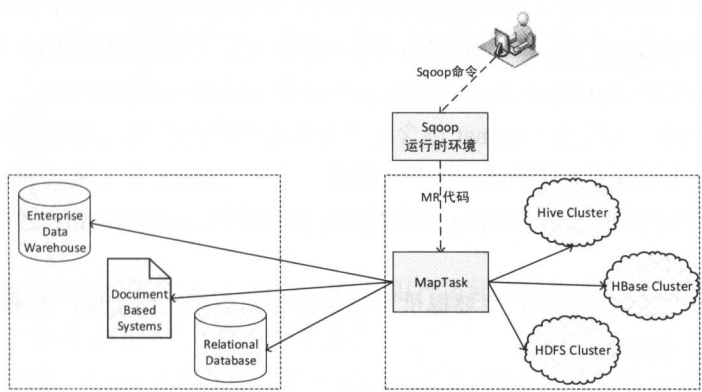


图 1-37 Sqoop 任务示例

1.2.10 序列化和持久化组件

处于两台不同机器上的进程间交换数据时，需要将数据序列化，以便于数据传输。一般，进程将其内存数据刷到硬盘上，称为数据持久化。大数据组件既要使用数据序列化技术，又要使用数据持久化技术。并且，一个高效的序列化或持久化工具，能大大加快数据存储与处理速度。

1. Avro^[55]：数据序列化系统

Avro 是一个序列化与 RPC 框架，其最初为 YARN 量身打造，不过考虑到稳定性，YARN 最终采用 Protocol Buffers 作为序列化工具。尽管如此，Avro 依旧是一个优秀的序

列化框架。Avro 核心思想是模式 (Schemas, 围绕 JSON 定义), 从功能上看, Avro 类似于 Thrift、Protocol Buffers, 不同之处在于 Avro 支持动态模式, 这点很适合数据模式不断变化的应用场景。

2. Gora^[56]: NoSQL 的数据持久化工具包

Gora 是个开源的对象关系映射 (Object Relational Mapping) 工具包, 它提供了内存数据模型和大数据持久化工具。从功能上看, Gora 类似于 Hibernate, 不同的是 Hibernate 实现的是 RDBMS 数据的持久化, 而 Gora 实现的是 NoSQL 数据库数据持久化。Gora 支持列数据、Key-Value 对数据、文档数据和 RDBMS 数据的持久化, 支持的数据库为 HBase、Cassandra、Accumulo 和 MongoDB。

3. Parquet^[57]: 可重复的嵌套列数据结构工具包

Parquet 由 Twitter 开发, 它是 Google Dremel 论文中提及的嵌套列数据结构 (Nested Schema to Columns) 的开源实现。正如论文中所述, 在存储底层, Parquet 实现了记录拆分和组装算法 (Record Shredding and Assembly Algorithm) 来实现嵌套列模型的高效压缩和存储。

Parquet 主要包含 parquet-format、parquet-mr 和 parquet-compatibility 三个包, 这三个包对应 Parquet 的三大功能。format 包含可重复嵌套列数据结构, 元数据的 Thrift 定义。mr 则提供了格式读取器、转换器, 这些转换器可实现 Hadoop 原生格式 parquet 嵌套列格式互转, 至于上层组件, 目前 Parquet 只支持 Pig 格式转换。compatibility 包则是利用 Thrift 提供其他语言支持。

Parquet 和上层组件无关, 主要提供可重复嵌套列数据结构实现, 未来计划为所有高层组件提供统一接口。

1.2.11 调试工具

调试 MapReduce 代码并不是件容易的事情, 编者写了三年的 MapReduce 代码, 经常发生单机下可以执行的代码到集群上不能执行, 单机下不能执行的到集群上却可以成功执行的情况, MRUnit 未出现之前, 编者需要到集群上反复调试代码。有了 MRUnit^[58], 用户在编写 MapReduce 代码时即可进行单元测试, 显然大大方便了开发与测试 MR 代码。

1.2.12 安全性组件

1. Knox^[59]: 统一的轻量级集群访问中间层

当在 Redhat 上装好 MySQL 后, 最基本的安全性措施是新增一个同名的 mysql 用户

和 mysql 组，此时对 MySQL 可采用如下权限分配方式：mysql 用户本身拥有最高权限，同组用户拥有读权限，不同组用户无任何权限（连读都不可以）。

和上述场景类似，集群建好后，默认情况下对每个服务都新增一个用户，如对 HDFS 服务会新建 hdfs 用户，YARN 服务对应 yarn 用户，Oozie 服务对应 oozie 用户。启动服务时，各个服务默认以自己的用户名启动。

由于大数据组件太多（本章就已讲述了 47 个），系统会创建大量用户，加之每个用户（如 yarn）都可以设置自己的用户名和密码，当外部 client 端 allen 用户访问集群时，使用 yarn 要输入 yarn 密码，使用 oozie 要输入 oozie 密码，显然非常不方便。此时即可借助 Knox，其在群集中为 Hadoop 服务提供 SSO（单点登录）和访问（图 1-38）。

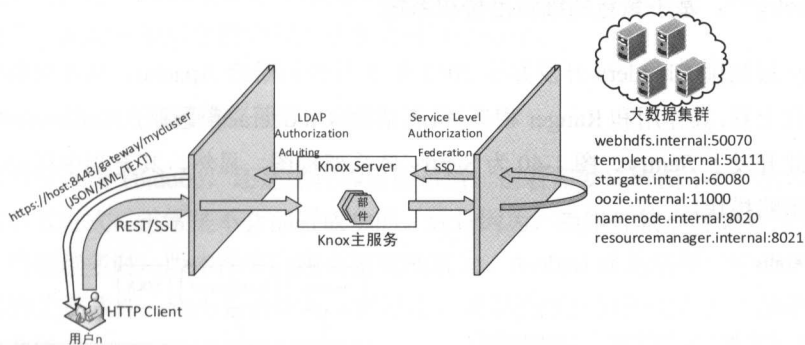


图 1-38 Knox 效果图

Knox 在控制访问权限的同时，可大大简化用户访问集群数据与服务的复杂性、访问群集数据和执行作业的用户登录操作的复杂性。

值得注意的是，Knox 还提供了云审计功能，通过该功能，服务提供商可按时或按量收费云服务。

2. Ranger^[60]：数据安全管理中心

由于大数据组件实在太多，且每个组件都有自己的用户名和密码，太不方便用户使用，Knox 旨在为 Client 提供单点登录服务（Single-Sign-On），让 Client 可以用一个账号一次登录却能使用所有组件，显然，Knox 是以 Client 为中心的，主要面向操作，其为 Client 屏蔽了集群内部一切登录认证。

和 Knox 不同，Ranger 则更加注重数据安全，它是 Hadoop 平台的数据安全管理中心，负责授权、监控和管理 Hadoop 平台的数据安全性（图 1-39）。

Ranger 起源于 Hortonworks，为提高 Hadoop 平台数据安全性，Hortonworks 开发了“Advanced Security”并将其捐赠给 Apache。开始时，Apache 将“Advanced Security”命名为 Argus，后来才重命名为 Ranger。和 Knox 不同，Ranger 首先拥有授权或否决前台 Client 访问权利；其次，它模拟 Linux 权限策略，设置此 Client 能够操作哪个文件。不过

目前 Ranger 只支持 HDFS、Hive、HBase、Storm、Knox、YARN、Solr 和 Kafka，支持的文件类型也仅为 Files、Folders、Databases、Tables 和 Column，后续过程中会开发更多支持包。

以 HDFS 为例，假设 HDFS 有文件 ainjupt.txt，用户 joe 预通过 WebHDFS 访问 ainjupt.txt。第一个问题是，joe 是否有权使用 WebHDFS？第二个问题才是 joe 是否有权查阅 ainjupt.txt？在此场景下，Ranger 首先有否决 joe 访问 WebHDFS 的权利；而当 joe 有权访问 WebHDFS 时（Ranger 授权），还可通过 Ranger 向 joe 授予不同的操作权限（如只允许 joe 读取 ainjupt.txt，不允许修改）。

3. Sentry^[61]：基于角色的细粒度授权系统

Sentry 最初由 Cloudera 开发并于 2013 年 5 月份捐赠给 Apache。从在集群中的功能与角色定位上看，其作用和 Ranger 相当。编者猜测，Cloudera 感受到了 Hortonworks Ranger 压力，因此开发了 Sentry。图 1-40 为 Sentry 体系架构图，显然，其充当中间层，相当于一个中间策略模块。

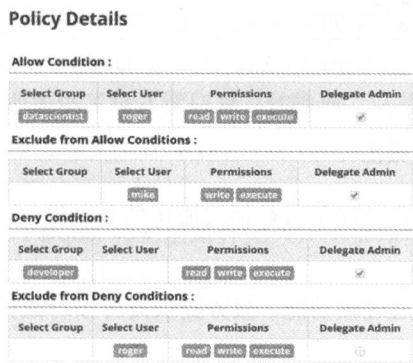


图 1-39 Ranger 权限分配示例（取自官网）

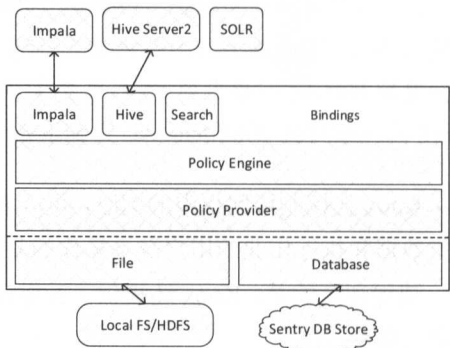


图 1-40 Sentry 体系架构

1.2.13 兼容性组件

根据编者经验，开源软件缺陷之一就是其存在兼容性问题。大数据涉及多达 47 个组件，显然，能将这些组件以完全兼容方式部署起来，本身就是一个巨大的挑战。

由于每个大数据组件都由不同组织开发，并且在开发过程中，组件自身依赖的底层组件可能也在不停升级，比如几乎所有组件都依赖 Hadoop，但 Hadoop1.0 与 2.0 在架构上几乎完全不同；再比如 HDFS 指定使用 Tomcat8，而 Oozie 却使用 Tomcat7，Kafka 使用 Jdk7，而 YARN 使用 jdk8，当这些组件集成到集群中后，兼容性问题会被一步步放大。Bigtop^[62]内置了用来规范组件间版本的一套机制，除此以外，Bigtop 还提供了打包功能，

如可使用 Bigtop 将 YARN 源码打包成 rpm 格式。

实际上，部署大数据组件的最大障碍就是版本问题，由于组件都来自不同的开发组以及组件之间依赖关系，想要彻底解决版本兼容性问题几乎是不可能的问题，在这种情况下只能尽量使用相互兼容的组件。

1.2.14 集群部署与管理组件

诚然，软件需要为用户提供实用功能，但软件自身的运行也是需要管理的，特别是一些大型应用软件（如 Oracle）。当大数据组件全部都运行时，如果手工管理这些组件，将非常烦琐，此时可使用集群管理工具来管理这些组件。

1. Ambari^[63]：大数据集群部署、管理、监控工具

如果仅仅部署 Hadoop，还是比较容易的事情，但若需要部署图 1-2 中的所有组件，且全采用手工方式，正常是不会部署成功的。这是因为，当组件较少时兼容性问题较小，但当组件越来越多时，版本冲突问题会被逐渐放大。Ambari 就是这样一个部署、管理和监控集群的工具软件，它内部维护着一套稳定、兼容的所有组件版本表。部署时它会自动选择配套版本。

当集群正常运行时，还可通过 Ambari 监控大数据集群，比如某服务发生故障时，Ambari 会自动向运维人员发送警报邮件。Ambari 使用 REST（轻量级 Web Service 架构）风格获取各个大数据组件状态与服务信息，为用户提供了轻量级编程接口（图 1-41）。

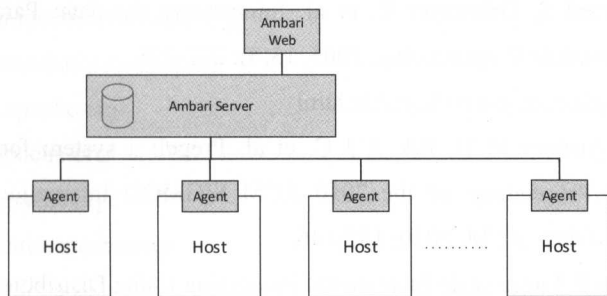


图 1-41 Ambari 体系架构

习 题

1. 请分层简述 Google 分布式架构。
2. 简述 Caffeine 整个处理流程。

3. 简述 Apache 下所有顶级项目及其主要功能。
4. 简述 Apache 下所有云项目及其主要功能。
5. 简述 Apache 下所有大数据项目及其主要功能。
6. 查阅相关资料, 实例演示面向行存储、面向列存储和嵌套列存储。
7. 何为增量处理机制? 处理增量数据的常见方法有哪些?
8. 简述 BSP 并行化模型。
9. 简述松耦合和紧耦合应用场景。
10. 对于如此多的大数据组件, 如何确保组件本身安全性, 又如何确保组件管理的数据的安全性。
11. 在大型集群中, 如何才能够统一管理这么多的大数据组件?
12. 在大型集群中, 部署所有大数据组件时, 最稀缺的系统资源是什么?

参考文献

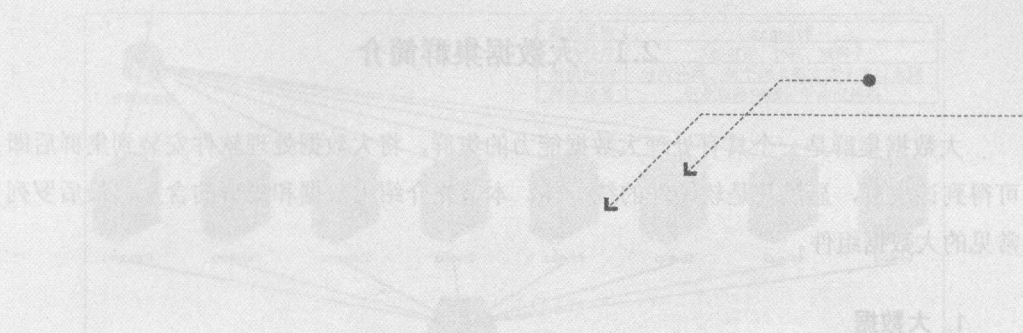
- [1] <http://research.google.com/archive/googlecluster.html>
- [2] <http://labs.google.com/papers/gfs.html>
- [3] <http://labs.google.com/papers/chubby.html>
- [4] <http://labs.google.com/papers/mapreduce.html>
- [5] Pike R, Dorward S, Griesemer R, et al. Interpreting the data: Parallel analysis with Sawzall[J]. Scientific Programming, 2005, 13(4): 277-298.
- [6] <http://labs.google.com/papers/bigtable.html>
- [7] Malewicz G, Austern M H, Bik A J C, et al. Pregel: a system for large-scale graph processing[C]//Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010: 135-146.
- [8] Peng D, Dabek F. Large-scale Incremental Processing Using Distributed Transactions and Notifications[C]//OSDI. 2010, 10: 1-15.
- [9] Baker J, Bond C, Corbett J C, et al. Megastore: Providing Scalable, Highly Available Storage for Interactive Services[C]//CIDR. 2011, 11: 223-234.
- [10] Melnik S, Gubarev A, Long J J, et al. Dremel: interactive analysis of web-scale datasets[J]. Proceedings of the VLDB Endowment, 2010, 3(1-2): 330-339.
- [11] Sigelman B H, Barroso L A, Burrows M, et al. Dapper, a large-scale distributed systems tracing infrastructure[J]. Google research, 2010.

- [12] <http://googleblog.blogspot.com/2010/06/our-new-search-index-caffeine.html>
- [13] Corbett J C, Dean J, Epstein M, et al. Spanner: Google's globally distributed database[J]. ACM Transactions on Computer Systems (TOCS), 2013, 31(3): 8.
- [14] <http://www.apache.org/>
- [15] <http://lucene.apache.org/>
- [16] <http://nutch.apache.org/>
- [17] <http://hadoop.apache.org/>
- [18] <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html>
- [19] <http://zookeeper.apache.org/>
- [20] <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/index.html>
- [21] <http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html>
- [22] <http://hbase.apache.org/>
- [23] <http://tez.apache.org/>
- [24] <http://pig.apache.org/>
- [25] <http://mahout.apache.org/>
- [26] <http://hive.apache.org/>
- [27] <http://tajo.apache.org/>
- [28] <http://crunch.apache.org/>
- [29] <http://datafu.incubator.apache.org/>
- [30] <http://mrql.incubator.apache.org/>
- [31] <http://kylin.apache.org/>
- [32] <http://lens.apache.org/>
- [33] <http://drill.apache.org/>
- [34] <http://accumulo.apache.org/>
- [35] <http://cassandra.apache.org/>
- [36] <http://phoenix.apache.org/>
- [37] <http://hama.apache.org/>
- [38] <http://giraph.apache.org/>
- [39] <http://storm.apache.org/>
- [40] <https://samoa.incubator.apache.org/>
- [41] <http://spark.apache.org/>
- [42] <http://reef.apache.org/>
- [43] <http://tinkerpop.incubator.apache.org/>

- [44] <http://ignite.apache.org/>
- [45] <http://flink.apache.org/>
- [46] <http://slider.incubator.apache.org/>
- [47] <http://twill.incubator.apache.org/>
- [48] <http://incubator.apache.org/blur/>
- [49] <http://oozie.apache.org/>
- [50] <http://falcon.apache.org/>
- [51] <http://flume.apache.org/>
- [52] <http://kafka.apache.org/>
- [53] <http://chukwa.apache.org/>
- [54] <http://sqoop.apache.org/>
- [55] <http://avro.apache.org/>
- [56] <http://gora.apache.org/>
- [57] <http://parquet.apache.org/>
- [58] <http://mrunit.apache.org/>
- [59] <http://knox.apache.org/>
- [60] <http://ranger.incubator.apache.org/>
- [61] <https://sentry.incubator.apache.org/>
- [62] <http://bigtop.apache.org/>
- [63] <http://ambari.apache.org/>

(1) 集群部署

一种就是由多台服务器组成，每台服务器上都安装HADOOP，然后将HADOOP的配置文件复制到每台服务器上，最后再在每台服务器上安装HADOOP的客户端，这样就能实现集群部署了。另一种就是由一台服务器组成，这台服务器上安装HADOOP，然后将HADOOP的配置文件复制到其他服务器上，最后再在其他服务器上安装HADOOP的客户端，这样也能实现集群部署了。这两种方法都可以实现集群部署，但是第一种方法比较麻烦，第二种方法比较简单。



第2章

大数据集群

HADOOP
BEING DIGITAL

大数据集群的部署方法有很多种，其中一种是使用虚拟机来部署。另一种是使用物理服务器来部署。还有一种是使用云服务来部署。每种方法都有其优缺点，需要根据实际情况来选择。

在部署大数据集群时，需要注意以下几点：1. 选择合适的硬件配置。2. 选择合适的操作系统。3. 选择合适的网络环境。4. 选择合适的部署方法。

(3) 部署管理

部署 5

部署管理是指对大数据集群的部署过程进行管理和控制。这包括对集群的规划、设计、部署、维护和升级等方面进行管理。

对大部分用户来说,大数据集群神秘莫测,难以捉摸,本章的目的就是构建这样一个属于自己的大数据集群。在给出大数据相关定义后,首先简单介绍云创大数据(<http://www.cStor.cn>)公司内部的大数据集群 bigCstor,然后以 bigCstor 为模板构建本书范例大数据集群 littleCstor。

2.1 大数据集群简介

大数据集群是一个具有处理大数据能力的集群,将大数据处理软件安装到集群后即可得到该集群,显然其是软硬件的统一体。本节先介绍大数据和集群的含义,最后罗列常见的大数据组件。

1. 大数据

大数据一词最早由著名咨询公司麦肯锡^[1]提出,在其报告《Big data: The next frontier for innovation, competition, and productivity》中,麦肯锡指出大数据是指大小超出常规的数据库获取、存储、管理和分析能力的数据集。进一步,IDC^[2]给出了大数据四大基本特征:

- 海量的数据规模 (Volume);
- 快速的数据流转和动态的数据体系 (Velocity);
- 多样的数据类型 (Variety);
- 巨大的数据价值 (Value)。

需要指出的是,并不是只有超过“PB 级”数据量的数据才能称为大数据,超出单机处理能力的数据也可称为大数据。比如某个包含十亿条记录的 1TB 数据,在实际处理时假如每条记录需要 10 毫秒处理时间,则单核服务器需要 115 天(2760 个小时)才能处理完,32 核 CPU 也需要近 4 天时间。基于此,亚马逊的大数据科学家 John Rauser 给出了一个更直观、更简单的定义:

大数据是指超过了任何一台计算机处理(包括存储和计算)能力的数据库^[3]。

在实际应用中,这个概念更加易于理解,本书中的大数据组件在物理部署上几乎都跨越多台机器,即用分布式方案来解决单机处理瓶颈。

2. 集群

集群指的是通过网络连接起来的多台服务器的集合(图 2-1)。

(1) 集群配置

通常情况下、集群应至少包含 10 台以上服务器，集群中单机配置请参考图 2-1 中的表格，和普通 PC 不同，服务器一般至少配备两个网卡，分别用于设置此服务器的内外网 IP。在配置服务时应根据服务性质的不同，选择不同的网络方式，比如集群内部交换数据时，应当使用内网。外网访问集群内某台机器服务时，应当使用外网。当然，在集群的硬件配置方面，没有严格限制，编者列举的是最常见情况。不过，在 OS（操作系统）方面，请尽量使用 Redhat^[4]或 CentOS^[5]。

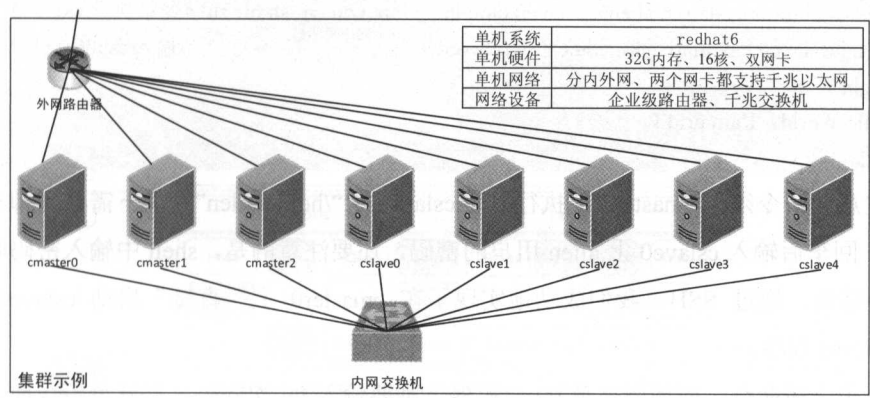


图 2-1 集群示意图

(2) 单机管理

所谓的管理单机指的是管理单机上某正在执行的后台进程，常见的管理包括启动进程、查看进程、关闭进程等。在管理本机服务时，一般会将服务启动、查看、关闭等命令写入一个或多个文件，我们称此类文件为脚本，常见的脚本由 Shell 或 Python 编写。

以 cslave0 为例，以下脚本完成在 cslave0 上启动包 HDPAction.jar 中 HelloWorld 程序：

```
echo "Begin"                                #打印字符串 Begin
java -cp /home/allen/HDPAction.jar njupt.HelloWorld  #Shell 方式启动 Java 程序
sleep 6                                         #休眠 6 秒
echo "End"                                     #打印字符串 End
```

保存并命名此脚本为 hw.sh，下述命令完成在 cslave0 上执行此脚本，后面三行为程序或脚本输出的内容。

```
[allen@cslave0 ~]$ sh hw.sh                #cslave0 上执行 hw.sh 脚本，下面为输出
Begin
Hello World, I'am Allen !
End
```

(3) 集群管理

所谓集群管理指的是同时管理集群内所有机器，比如同时给上千台服务器执行一个命令，拷贝一个文件，杀一个进程等。由于集群由多台机器组成，集群管理必然涉及机

器之间相互访问。称从一台机器登录到另外一台机器为远程登录，常见的远程登录工具为 SSH^[6]。

涉及集群管理时，可将集群中机器分为管理者和被管理者两个角色。一般情况下，集群管理都是从管理机器上远程登录被管理机器，然后启动被管理机器上相关脚本。目前最常用的远程登录工具为 SSH。具体实现时，首先在管理节点上 SSH 登录被管理节点，然后执行被管理节点上的 shell 脚本。下面的命令完成从 cmaster0 以 allen 用户远程登录 cs slave0 并执行 cs slave0 上的 hw.sh 脚本。

```
[allen@cmaster0 ~]$ ssh allen@cs slave0.cloudlab.njupt.edu.cn 'sh hw.sh' #登录执行 cs slave0 上脚本
allen@cs slave0.cloudlab.njupt.edu.cn's password: #输入 cs slave0 机 allen 密码
Begin
Hello World, I'am grid !
End
```

注意此命令须在 cmaster0 上执行，而 cs slave0 上“/home/allen”目录下需要有脚本文件 hw.sh。回车后输入 cs slave0 上 allen 用户的密码，还要注意的，shell 中输入密码时，光标不会移动。通过 SSH，我们成功地实现了在 cmaster0 上“直接”启动 cs slave0 上的 HelloWorld 程序。

对于上述命令，假如要登录 99 台机器，则要输入 99 次密码，显然很不合理。通过配置 SSH，还可实现 cmaster0 到 cs slave0 无密钥登录，从而避免输入密码的情况，更加方便集群的统一管理。进一步，假定集群中有一台管理机，99 台被管理机，实现管理机到被管理机无密钥登录后，可以在管理机上统一管理集群中所有机器。需要指出的是，在小型的集群管理中可以使用 for 循环来组织 SSH，不过在大型集群中，由于 for 循环性能不佳且不是同步并行执行，会带来莫名其妙的问题，此时编者建议使用 pssh、pdsh、mussh、cssh、dsh 等工具。

简单地说，SSH 和 QQ 等程序并无本质区别，其都是建立在应用层上的用户程序。不同的是，它们提供的功能完全不同，SSH 主要功能为远程登录和数据加密。通过 SSH，可以实现从 cmaster0 远程登录到 cs slave0，且在传输过程中，用户名和密码已被加密；还可实现 cmaster0 和 cs slave0 之间相互传输数据，当然这个数据也已被加密。SSH 还有其他功能，这里不再详述，关于 SSH 无密钥登录，请读者参考第 3 章。

在单机环境下，SSH 一般用不到；在集群环境下，SSH 主要用于从管理机远程登录被管理机，实际管理（启动、查看、关闭）本机服务时，必须借助本机上 shell 或 python 脚本，当然可将 SSH 命令和管理命令写入同一个脚本。总之，在集群中 SSH 主要用于安全的无密钥（需配置）远程登录。

3. 大数据组件及套件

大数据组件指的是用来处理大数据的软件。显然，处理大数据的软件不止一个，称多个大数据组件为大数据组件集或大数据套件。

Apache（Apache Software Foundation，Apache 软件基金会）是当今世界最有名的开源软件组织。Apache 下维护着众多世界顶级开源项目，有的项目由 Apache 自主开发，有的则是第三方组织赠予 Apache，比如 Hadoop、ZooKeeper 等，都是 Apache 自主研发的项目，而 Spark 由伯克利大学贡献，Hive 由 FaceBook 贡献，Storm 则由 Twitter 捐赠。Apache 以 Hadoop 为基础，直接或间接维护着近 50 多个 Hadoop 生态圈项目（图 2-2）。

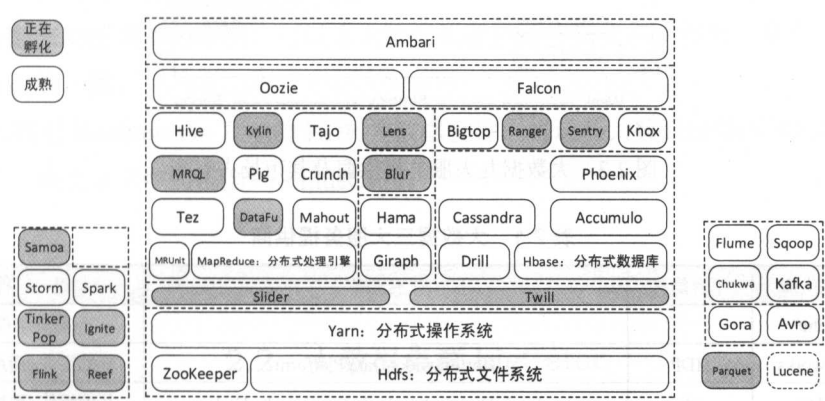


图 2-2 Apache 大数据组件

不过这 50 多个组件中有些组件功能类似（如 Hive 和 Tajo），有些组件正逐步淘汰，有些则正在研发（图中阴影组件）。在这些组件中，最常见的为如下 15 个，这些组件功能各异，相互配合完成大数据处理（存储、转换、分析等）：

HDFS、MapReduce2、YARN、Tez、HBase、Sqoop、Oozie
Pig、Hive、ZooKeeper、Storm、Flume、Kafka、Slider、Spark

为区分下文的“商用版”大数据套件，称这 15 个常用大数据组件集为 **Apache 社区大数据套件**。和社区大数据套件相对应，称各商业公司发行的大数据组件集为**商用大数据套件**。

当前大数据解决方案提供商主要为 Hortonworks^[7]、Cloudera^[8]、MapR^[9]、AWS^[10]、IBM^[11]、Microsoft^[12]、Pivotal^[13]、Intel^[14]和 Teradata^[15]（图 2-3）。

值得注意的是，这九大服务提供商解决方案的底层基础都是**社区大数据套件**。不同的是，各大公司高层产品各有特色，且一般这些高层产品为商用组件，比如 Hortonworks 的 Ambari，Cloudera 的 Impala^[16]，MapR 的 MapR-DB^[17]，这些高层产品大多与 NoSQL、BI 或集群管理相关。我们可以理解为，底层产品直接使用**社区大数据平台**，高层产品则

由各大公司独立研发（表 2-1）。

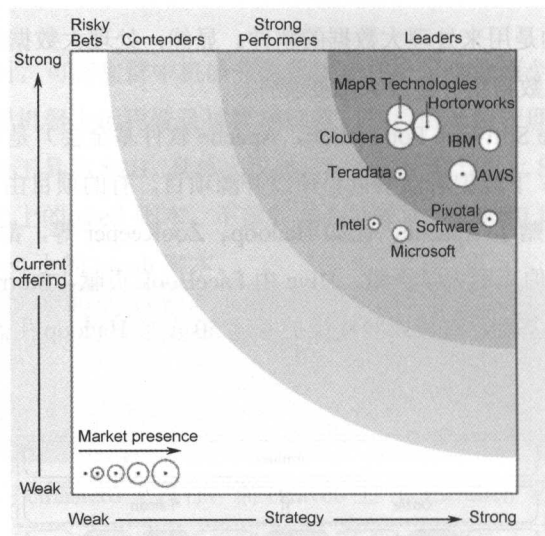


图 2-3 大数据九大服务提供者及其市场占有率

表 2-1 大数据三大服务提供商

方案提供商	平台简称	平台全称	组 件
Apache			基础组件
Hortonworks	HDP	Hortonworks Data Platform	基础组件+商用组件
Cloudera	cdh	Cloudera's Distribution Including Apache Hadoop	基础组件+商用组件
MapR	mdp	MapR Data Platform	基础组件+商用组件

由于各大公司的商用大数据套件都依托于 Apache 社区大数据套件，故除了介绍三大主流发行版外，我们也将 Apache 社区版罗列在表格内。由于编者仅有 HDP 和 cdh 长期运维与开发经验，对于 mdp 编者只知其 HDFS（MapR 对其进行了重构）优于其他所有发行版，故关于“Hdp、Cdh 和 Mdp 这三大发行版各自优缺点”，编者不便评述。

不过，由于这三大公司内商用组件的开源程度不同，Apache 对其发行版认可度也不尽相同，本书中选用 Apache 认可度最高、开源性最好的 HDP，即 Hortonworks 公司发行版 HDP。

从各大公司对社区大数据套件的包装可以看出，产业界迎来了一次技术浪潮。以 Linux 为例，自从 1991 年 Linux 内核公布后，各大公司在内核基础上开发了相应的商用软件，发行时，这些公司将开源内核和自主研发的商用软件一起打包发行，这些不同的发行版即称为 Linux 发行版，如 Redhat、Ubuntu^[18]、Fedora^[19]、Debian^[20]等都是不同的 Linux 发行版。这些发行组织大多都通过商用软件、培训服务、增值服务、升级服务等盈利。

和 Linux 发行版商业模式相同，各大商业公司在社区大数据套件基础上开发高层组件，发行时将高层组件和底层社区版大数据套件一起打包发行，这就形成了当前诸多大数据平台发行版（如 HDP、cdh、mdp 等），各大公司一般通过自身发行版里的商用软件、培训服务、增值服务、升级服务等方式盈利。

4. 大数据集群

在集群上部署一整套大数据软件后的集群称为大数据集群。显然，大数据集群是软硬件的统一体。

请读者注意，没有集群，可以说大数据组件只是一堆无用的代码，同样，没有大数据组件，集群就是单机的累加，依旧没有能力处理海量数据。单个组件功能有限，多个组件组合起来功能则更为全面，可以说大数据集群是硬件集群和软件集（多个大数据组件）的有机统一体。

本书选用 Hortonworks 发型的 HDP 大数据套件，将这些大数据组件部署到编者的 9 台单机后，就是本书中的范例——littleCstor。

2.2 大数据集群 bigCstor

作为国内领先大数据和云计算服务提供商，在云计算的基础设施方面，cStor^[21]使用 Docker^[22]来提供公有云和私有云服务；在大数据套件层面，cStor 选用 Hortonworks 的 HDP 大数据套件。HDP（Hortonworks Data Platform）集成和优化了各种开源大数据系统，为企业提供可用的大数据平台，让组织能够采用现代化数据架构。HDP 以 YARN 作为其架构中心，是一系列处理方法（从批量到交互式再到实时）的多工作负荷数据处理平台。

在 cStor 内有一个很大的大数据集群（图 2-4），内部称此集群为 bigCstor。在 bigCstor 上，不仅部署了一套完整的 HDP，还部署了 Redis^[23]、LevelDB^[24]和图 2-2 中的几乎所有组件。集群暂不对外开放，不过集群的物理拓扑（图 2-4）可以向读者展示。

需要指出的是，在 cStor 内部，bigCstor 实际上是部署在云上的。就部署过程而言，在云端部署 HDP 和在真实集群上部署 HDP 的确不同，当选择在云端部署 HDP 时，我们需要用到 Cloudbreak^[25]，整个部署过程较复杂，而在真实集群上部署 HDP 则简单许多。不过在产品使用方面，无论是云端的大数据集群还是真实机器上的大数据集群，二者在使用上并无任何区别。就如何将大数据套件 HDP 部署到云端，有兴趣的读者可参考 Cloudbreak、Docker，云计算和大数据的结合应该是未来几年的研究重点。

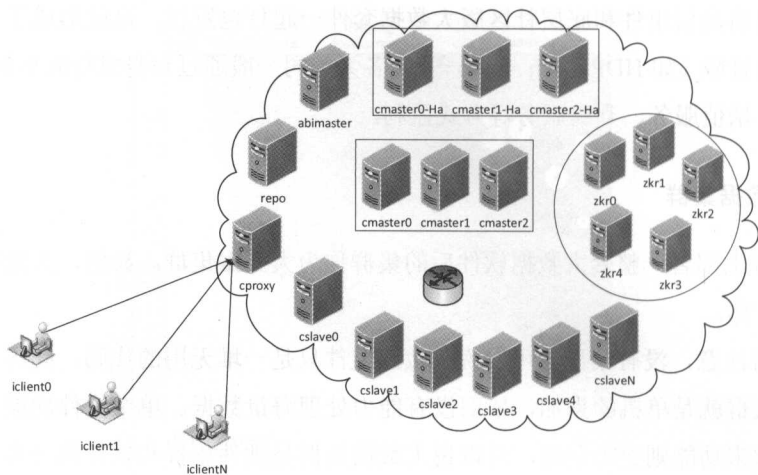


图 2-4 bigCstor 拓扑图

下面从单机配置、各机角色定位和已部署的大数据组件这三个方面简单介绍 bigCstor。

1. bigCstor 单机配置

bigCstor 集群采用当前最标准的硬件和 OS 配置，表 2-2 为 bigCstor 主要的硬件和 OS 参数。

表 2-2 bigCstor 硬件和 OS 配置

单机系统	redhat6
单机硬件	32G 内存、16 核、双网卡
单机网络	双网卡、分内外网
网络设备	企业级路由器、企业级千兆交换机
JDK 版本	jdk-7u67-linux-x64.tar.gz

2. bigCstor 单机角色定位

在 bigCstor 内，由于组件太多，cmaster 本身被拆分成了 3 个节点 (cmaster0、cmaster1、cmaster2)；cmasterX-Ha 是这三个节点的 HA 节点；abimaster 上为 Ambari 主节点；ZooKeeper 集群主要提供 ZooKeeper 服务；cslaveX 为从节点；cproxy 为代理节点，代理一些组件主服务的 Web 功能，从而转移外来的恶意 Web 攻击；repo 节点是集群仓库节点，我们规定集群中所有节点下载任何组件时，都自动到 repo 下载，这样就能确保集群中所有组件版本一致。此外，repo 节点上还运行时钟滴答服务，来校准集群内各机时钟。

iclinetX 节点本身并不属于集群，它们只是一个客户端，理论上客户节点可以是世界上任何一台机器，表 2-3 为 bigCstor 内各机角色定位表。

表 2-3 bigCstor 内各机角色定位表

定 位	角 色	机 器
仓库	仓库节点	repo
集群	代理节点	cproxy
	Ambari 主节点	abimaster
	主节点	cmaster0、cmaster1、cmaster2
	主节点热备节点	cmaster0-Ha、cmaster1-Ha、cmaster2-Ha
	从节点	cslave0~cslaveN
	ZooKeeper 节点	zkr0~zkr4
客户端	客户端节点	iclient0~iclientN

从某种意义上说、bigCstor 被规划的很复杂，这主要是因为：

- 集群本身较大（400+节点数）；
- 提高服务可靠性；
- 明确单机角色定位，从而简化大集群拓扑图；
- 方便集群横向扩展。

比如，为提高 ZooKeeper 可靠性，ZooKeeper 集群本身就由五台机架式服务器（刀片）组成；为提高主服务的高可靠性，借助 ZooKeeper，针对每台 cmaster 都部署了其相应的 HA（High availability）节点。集群中的 repo 主要是为了确保集群内组件版本一致性，cproxy 则用于规避部分 Web 攻击。

3. bigCstor 上部署的大数据组件

bigCstor 上部署了 Hortonworks 的 HDP，此 HDP 版本号为 HDP-2.2.6.0-2800，该 HDP 包含如下组件。

HDFS、MapReduce2、YARN、Tez、Hbase、Sqoop、Oozie、Flacon、Metrics
Pig、Hive、ZooKeeper、Storm、Flume、Kafka、Slider、Spark、Ranger、Knox

此外 cmaster0-Ha、cmaster1-Ha、cmaster2-Ha 这三个节点上，还部署了 Redis、LevelDB 和 MySQL，这几个数据库是实现大数据实时处理的必备组件，其应用领域甚至比大部分的大数据组件还要广阔。

本书中选用真实的机器来部署本书范例集群 littleCstor，站在用户角度上，大数据组件到底是部署在云端还是真实集群上，用户使用时并无区别，由于本书是专门讲解大数据集群的专业书籍，并不涉及云计算和大数据交叉学科，为规避复杂性，编者使用真实集群部署大数据组件。

2.3 我的大数据集群 littleCstor

参照 bigCstor，编者使用图 2-5 中的硬件，部署了一个属于编者自己的大数据集群 littleCstor（图 2-6），请注意 littleCstor 是本书的示例集群，本书中后续章节都会按照 littleCstor 来讲解。

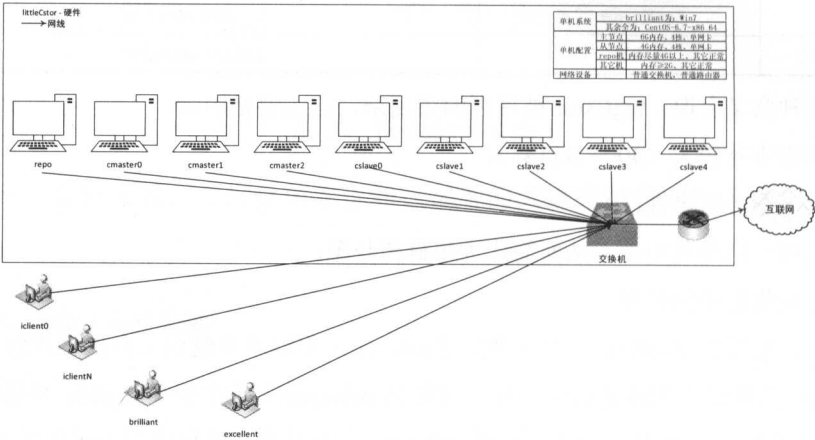


图 2-5 littleCstor 硬件拓扑图

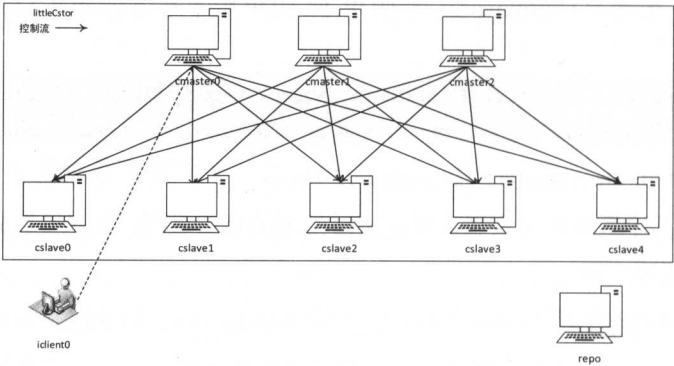


图 2-6 littleCstor 拓扑图

为了和 bigCstor 对应，编者称此集群为 littleCstor。为保持集群拓扑清晰性，在损失

可靠性的情况下,通过将包含 ZooKeeper 在内的所有组件主服务均匀部署在 3 台 cmasterX 上,从而去除了一些不必要的节点。读者所在机构或组织若要部署大数据集群,可先行参考 littleCstor,成功后再参照 bigCstor 部署自己的“bigCstor”。若要将“littleCstor”或“bigCstor”部署到云端,请参考 Cloudbreak 和 Docker。

由于需要部署的大数据组件太多,对 littleCstor 内单机配置有一定要求,下面讲述 littleCstor 内各机软硬件配置并给出部署过程中最低硬件配置。

1. littleCstor 硬件配置

表 2-4 为编者搭建 littleCstor 过程中单机和交换机主要参数。显然, littleCstor 硬件配置较低,不过此配置完全满足搭建需求以及本书中所有示例代码,用户在搭建大数据集群时,单机硬件配置不得低于表 2-4 中的参数。

表 2-4 littleCstor 硬件和 OS 参数

单机系统	CentOS-6.7-x86_64
单机硬件	6G 内存、4 核、单网卡
单机网络	单网卡
网络设备	普通交换机
JDK 版本	jdk-7u67-linux-x64.tar.gz

CentOS 由 Red Hat Enterprise Linux 依照开放源代码规定释出的源代码编译而成,其具有开源、免费、高可靠性等诸多优点, littleCstor 中我们选用 CentOS-6.7-x86_64。

2. 单机最低硬件要求

根据编者经验,即使普通 PC,其网卡和 CPU 也完全满足 littleCstor 搭建需求,但是当 littleCstor 上启动各个大数据组件时,各进程对内存量的需求非常大,故最低配置一般特指内存,表 2-5 为不同角色单机最低配置,低于这个配置,不可能部署成功。

表 2-5 littleCstor 硬件最低配置

定 位	角 色	内存最低配置
仓库	仓库节点	最好 4G 或 4G 以上
集群	主节点	单机内存最低 6G
	从节点	单机内存最低 4G
客户端	客户端节点	单机内存最低 2G

3. littleCstor 各机角色定位

littleCstor 内三个 cmaster 将会均匀地部署所有大数据组件的主服务,从节点则部署所有大数据组件从服务。客户端和仓库节点都不属于集群,但这两者都是必需的,这是

因为对用户来说，如果没有仓库节点，则不可能部署成功，表 2-6 为 littleCstor 内各机角色定位表。

表 2-6 littleCstor 内各机角色定位表

定 位	角 色	机 器	系 统
仓库	仓库节点	repo	CentOS-6.7-x86_64
Ambari	Ambari 主节点	cmaster0	
集群	主节点	cmaster0、cmaster1、cmaster2	
	从节点	cslave0、cslave1、cslave2、cslave3、cslave4	
	客户节点	iclient0	Win7
SSH 客户端	远程操作机	excellent	
		brilliant	

虽然集群中任何一台机器都是一个特殊的客户端，但为保证集群拓扑清晰性，编者使用单独机器 iclient0 作为客户机。正如表 2-6 所列，集群包含主从 8 个节点；客户端仅有 iclient0 一个节点；在集群正常运行时并不需要 repo 节点，repo 作用范围为集群搭建、添加节点（包括主、从、客户端）。此外，由于 cmasterX、cslaveX、repo、iclientX 都只开启了服务器模式，无桌面，编者有时须借助自身 PC（机器名为 brilliant，笔记本电脑，Win7 系统）远程登录至上述各机，excellent 为编者另一台 CentOS 笔记本，有时也使用它远程登录集群。不过集群本身仅为 cmasterX、cslaveX、iclientX 各机，不包括 brilliant、excellent，之所以有这两台机器是为了方便操作。

4. littleCstor 上的大数据组件

显然，和 bigCstor 相比，littleCstor 之所以被称为 little，是因为集群节点数量少，而非组件数量少。

littleCstor 上部署的是 Hortonworks 的 HDP，其完整版本号为 HDP-2.2.6.0-2800，此版本的 HDP 包含如下大数据组件：

HDFS、MapReduce2、YARN、Tez、HBase、Sqoop、Oozie、Flacon、Metrics
Pig、Hive、ZooKeeper、Storm、Flume、Kafka、Slider、Spark、Ranger、Knox

也就是说，littleCstor 上已经成功地部署了所有大数据组件，可以直接用来处理大数据。下面给出 littleCstor 的三幅效果图（图 2-7～图 2-9）。

图 2-7 为登录图，图中的①给出了登录网址，②提示用户输入用户名和密码，然后点击登录，登录后的效果如图 2-8 所示。

图 2-8 为集群控制面板图，它是登录后默认界面，图中①为集群名称 littleCstor，②罗列了 littleCstor 上当前部署的所有组件，③显示了当前登录用户名，④则为当前所在位置，即控制面板，⑤显示了集群状态信息，⑥提示集群中当前有两个警告消息。至于这

两个消息具体是什么，我们在图 2-9 中解释。

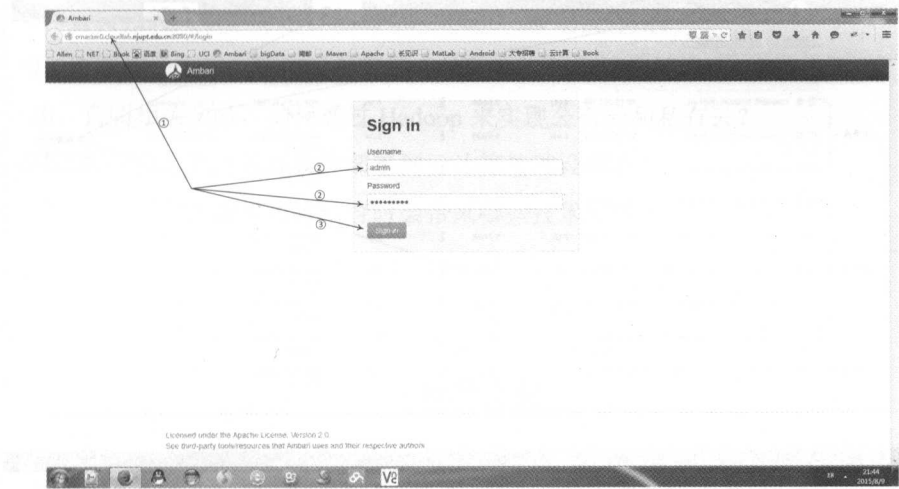


图 2-7 登录 littleCstor

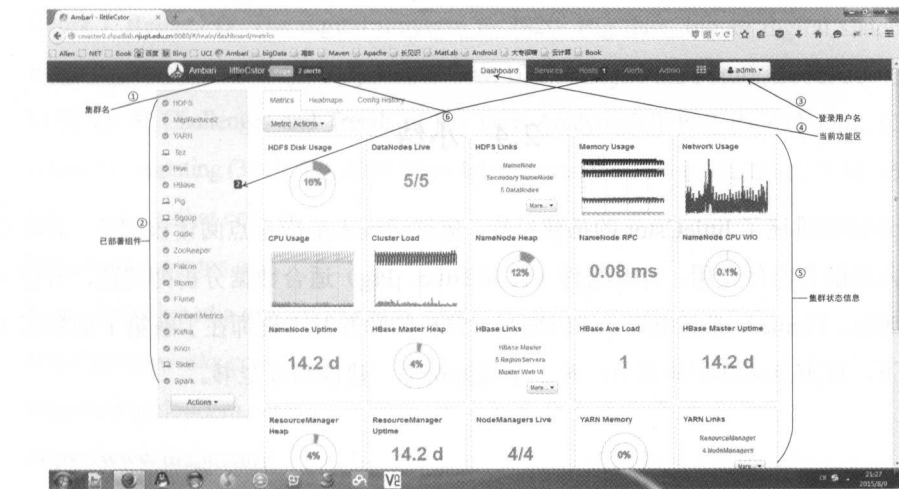


图 2-8 littleCstor 控制面板

从图 2-9 中可以看出，包括 iclient0 在内，集群共包含九个节点，其中有三个主节点，分别为 cmaster0~2，五个从节点，分别为 cslave0~4，最后是一个客户端节点 iclient0。littleCstor 的警告信息是由于 cslave2 内存太小引起的，由于 cslave2 内存只有 1.83G，HBase 不能成功启动，这就是编者一再强调 slave 机内存最低为 4G 的原因。

用户在构建大数据集群时，应为主节点配置 6G 以上的内存，从节点配置 4G 以上内存，低于此配置时，大数据集群不可能搭建成功。

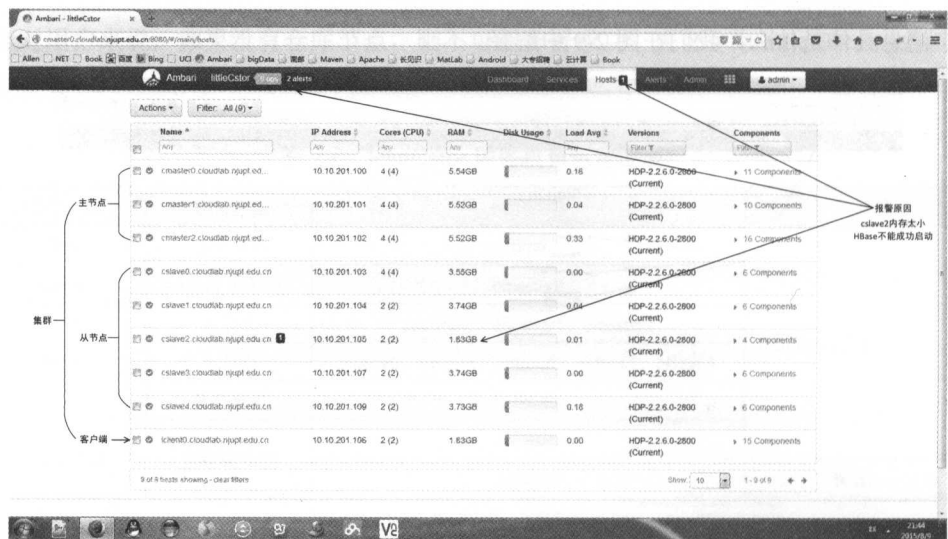


图 2-9 littleCstor 机器统计图

2.4 小结

本章详细讲述了 littleCstor 的搭建过程，集群运维人员应重点阅读该部分。后续章节则会深入讲解各组件使用，有些组件（比如 Hive、Pig）适合数据分析师阅读，有些（如 MapReduce、Flume）则更加侧重代码编写，分析师和开发工程师在了解第 1 章和第 2 章的情况下，可重点阅读相关章节；对于系统架构师，建议阅读全书。

习 题

1. 简述大数据相关定义。
2. 简述开源大数据平台、商用大数据平台及其区别与联系。
3. 集群环境下，一般使用什么工具实现机器之间无密钥登录？其变体又有哪些？
4. 简述为何选用 Redhat 体系的 CentOS 而不是 Ubuntu 来部署大数据组件。
5. 请查阅相关文档，对比九大“大数据服务提供商”底层技术异同点。
6. 简述 littleCstor 上部署了哪些大数据组件。既然第 1 章有那么多组件，为何不全部部署？

7. 请简述 littleCstor 中各机角色定位及其最低硬件配置。
8. 查阅相关文献, 简述 Docker 和 Cloudbreak 功能作用。
9. 查阅相关文献, 简述云计算核心技术、大数据核心技术。
10. 查阅相关文献, 如何通过 Hadoop 来实现公有云和私有云?
11. 如何将大数据处理软件部署到云计算基础设施上?
12. 云计算和大数据技术结合时会带来哪些技术变革?

参考文献

- [1] Manyika J, Chui M, Brown B, et al. Big data: The next frontier for innovation, competition, and productivity[J]. 2011.
- [2] Gantz J, Reinsel D. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east[J]. IDC iView: IDC Analyze the Future, 2012, 2007: 1-16.
- [3] Li H, Lu X. Challenges and Trends of Big Data Analytics[C]/P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2014 Ninth International Conference on. IEEE, 2014: 566-567.
- [4] <http://www.redhat.com/en>
- [5] <https://www.centos.org/>
- [6] https://en.wikipedia.org/wiki/Secure_Shell
- [7] <http://hortonworks.com/>
- [8] <http://www.cloudera.com/>
- [9] <https://www.mapr.com/>
- [10] <https://aws.amazon.com/big-data/>
- [11] <http://www.ibm.com/big-data/us/en/>
- [12] <http://www.microsoft.com/en-us/server-cloud/solutions/data-warehouse-big-data.aspx>
- [13] <http://pivotal.io/>
- [14] <http://www.intel.com/content/www/us/en/homepage.html>
- [15] <http://www.teradata.com/>
- [16] <http://www.cloudera.com/products/apache-Hadoop/impala.html>
- [17] <https://www.mapr.com/products/mapr-db-in-Hadoop-nosql>
- [18] <http://www.ubuntu.com>
- [19] <https://getfedora.org/>

- [20] <http://www.debian.org/>
- [21] <http://www.cstor.cn/>
- [22] <http://www.docker.com/>
- [23] <http://redis.io/>
- [24] <http://leveldb.org/>
- [25] <http://zh.hortonworks.com/Hadoop/cloudbreak/>

在 Hadoop 生态系统中，Hadoop、Hive、Mahout、Pig、Tez、ZooKeeper 等组件，都是运行在 Linux 操作系统上的。如果这些组件都运行在 Linux 操作系统上，那么它们之间的兼容性就很好了。

如果仅仅是 4 个组件，那无论组件之间如何冲突，我们都能找到解决办法。但是，如果组件的数量达到 20 个，那本冲突就会进一步复杂化。比如，如果组件之间冲突，那我们该怎么办？比如，如果组件之间冲突，那我们该怎么办？比如，如果组件之间冲突，那我们该怎么办？

如果仅仅是 4 个组件，那无论组件之间如何冲突，我们都能找到解决办法。但是，如果组件的数量达到 20 个，那本冲突就会进一步复杂化。比如，如果组件之间冲突，那我们该怎么办？比如，如果组件之间冲突，那我们该怎么办？比如，如果组件之间冲突，那我们该怎么办？

2) 集群管理工具

目前大数据的部署方式分为手工方式和工具方式。手工方式就是手动部署，工具方式就是使用集群管理工具。

第3章 集群管理工具 Ambari

HADOOP
BEING DIGITAL

在 Hadoop 生态系统中，Hadoop、Hive、Mahout、Pig、Tez、ZooKeeper 等组件，都是运行在 Linux 操作系统上的。如果这些组件都运行在 Linux 操作系统上，那么它们之间的兼容性就很好了。

图 3-1 Ambari 和 Cloudera Manager 对比

Cloudera Manager	Ambari	特点	适用场景
商业产品	开源产品	功能强大，支持多种组件	企业级部署
支持多种组件	支持多种组件	支持多种组件	支持多种组件

Cloudera 公司开发了 Cloudera Manager，并提供了配套的工具。Cloudera Manager 是一个非常强大的集群管理工具，它支持多种组件的部署和管理。Ambari 是一个开源的集群管理工具，它支持多种组件的部署和管理。

Ambari 是当前最常用的大数据集群管理工具。本章重点讲述使用 Ambari 管理 HDP, 并在 littleCstor 上进行实际操作。

3.1 Ambari 简介

所谓的管理大数据集群指的是部署大数据集群、启动大数据集群、监控大数据组件、查看大数据组件运行状态和关闭大数据组件, 即部署、启动、监控、查看和关闭。Ambari 是一款优秀的大数据套件管理工具, 不过它并不是唯一的, 在讲述 Ambari 之前, 编者先简单介绍当前主流的大数据平台管理工具。

1. 大数据套件管理工具

用户可采用手工或工具方式管理(部署、启动、监控、查看和关闭)大数据集群, 手工方式指的是纯手工管理大数据集群, 工具方式指的是使用第三方工具管理大数据集群。

当采用手工方式时, 以部署为例, 该方式整个过程均须用户执行, 细节太多, 且当涉及多个组件时, 用户须自己解决组件间版本兼容问题。

当前最主流的第三方管理工具为 ClouderaManager 和 Ambari, 它们分别由 Cloudera 和 Hortonworks 公司主导开发, 在市场占有率方面, 这两大公司几乎占到整个大数据市场份额的 70%^[1]。

1) 大数据集群部署难点

和其他软件不一样, 大数据集群搭建难度非常大, 从部署规划到最终成功搭建大数据集群, 步骤繁多复杂, 非专业人员即使参考文档部署成功了, 也可能在运行过程中出现这样或那样的问题, 这主要是由于:

- 部署规划不清;
- 集群环境复杂;
- 对组件自身认识不够;
- 组件太多;
- 组织之间版本不兼容。

简单地说, 编者现在需要在本人 PC (Windows 7 系统) 上安装 QQ, 这的确非常简单, 可大数据组件安装时不是装在一台 PC 上, 而是要装到多台 PC 上。进一步, 编者又

在 PC 上安装了 Eclipse，QQ 和 Eclipse 之间无任何联系，无须考虑版本兼容性问题，可当安装另一个大数据组件时，两者相互依赖，极有可能发生版本不兼容问题。

如果仅是三四个组件，那无论组件之间如何冲突，我们总能找到相互兼容的版本，可是当要部署的组件多达 20 个，版本冲突将会一步步被放大；集群环境和普通环境不同，比如集群中有些单机的 OS 是 CentOS，有些则是 Ubuntu，则肯定不能成功部署。即使全都是相同的 CentOS 系统，当集群中存在 CentOS-6.7 和 CentOS-6.5 时，部署依旧不会成功。这是由于 6.5 存在某特定 Bug，若要部署大数据平台，须先修复后再部署，而 6.7 已修复了此 Bug。当我们使用 Ambari 统一部署时，极有可能忽略此类小问题，最终导致一些奇怪问题。有问题就有解决方案，对于兼容性问题，编者推荐使用第三方工具 Ambari 自动解决版本冲突问题，对于系统 Bug 问题，编者建议按照官方文档，采用最标准的 redhat6 或 CentOS-6.7，请读者按文档严格使用标准环境。

2) 大数据集群管理工具

目前大数据集群管理方式为手工方式和工具方式，不过主流方式是使用专业工具 Ambari 或 ClouderaManager 来管理（部署、启动、监控、查看、关闭）大数据集群，以部署为例完全采用手工方式部署整个大数据套件几乎不可能成功。不过，采用手工方式部署大数据集群时，有利于部署者清晰理解集群物理拓扑，并加深对组件本身的认知。但由于手工方式太过复杂，加之组件间兼容性问题，可行性不高。

表 3-1 大数据两大部署方式

	手工方式	工具方式
难易度	难，几乎不可能成功	简单、可行
兼容性	自己解决组件间兼容性问题	自动安装兼容组件
组件支持数	支持全部组件	支持常用组件
优点	对组件和集群理解深刻	简单、容易、可行
缺点	太复杂，不可能成功	屏蔽太多细节、妨碍对组件理解

和手工部署方式相比，使用 Ambari 或 ClouderaManager 则会简单许多，这两个工具是当前大数据平台主流管理工具（表 3-1）。

表 3-2 Ambari 和 ClouderaManager 对比

工具名	所述机构	开源性	社区支持性	易用性、稳定性	市场占有率
ClouderaManager	Cloudera	商用	不支持	易用、稳定	高
Ambari	Hortonworks	开源	支持	较易用、较稳定	较高

Cloudera 公司开发了 ClouderaManager 并凭此取得了巨大成功，随后 Hortonworks 开发了 Ambari。实际上，ClouderaManager 和 Ambari 提供的功能类似（表 3-2）。在功能、易用性、稳定性与组件支持数量方面，Ambari 的确比 ClouderaManager 要稍显逊色，但

是作为 Apache 社区指定的管理工具，加之 Hortonworks 对 Ambari 的大力支持，本书使用 Ambari 来管理（部署、启动、监控、查看、关闭）littleCstor 集群。

2. Ambari 简介

Ambari 是由 Hortonworks 开发并赠予 Apache 的一栈式大数据集群管理工具。它主要面向大数据集群运维（管理）人员，通过 Ambari 提供的统一 Web 界面，集群管理员能够在 Web 界面上实现大数据集群部署、启动、监控、查看和关闭等一系列操作。

1) 物理拓扑

Ambari 采用 master/slave 架构，主节点运行 master 进程 AmbariServer，从节点运行 slave 进程 AmbariAgent。运行时，本机上的 AmbariAgent 负责监管本机上的大数据组件，中心机的 AmbariServer 则负责收集、汇总并显示集群中所有 AmbariAgent 发过来的“心跳”信息，图 3-1 为 Ambari 典型物理拓扑。

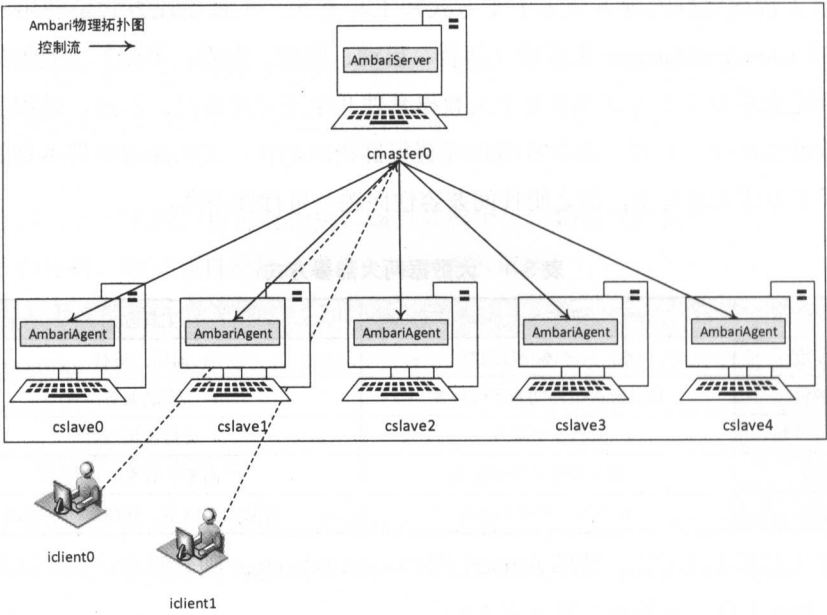


图 3-1 Ambari 典型物理拓扑

cmaster0 为主节点，负责收集、汇总并显示集群中所有 AmbariAgent 发过来的“心跳”信息。此处的显示主要为 Web 界面，即客户端（iclientX）可以通过 Web 查询集群状态信息。cslave0~4 为从节点，负责监控本机上的大数据组件。

2) 体系架构

Ambari 的 AmbariServer 采用 Java 编写，而 AmbariAgent 则由 Python 编写，之所以

这样设计是因为这两个模块面向的对象不同。对外 AmbariServer 需要向用户提供 Web 访问接口，对内 AmbariServer 需要实时接收 AmbariAgent 心跳信息。AmbariAgent 功能更加特别，对内它主要负责管理（开启、监控、查看和关闭）本机诸多大数据进程，对外它通过读取本机大数据进程输出的日志信息，向 AmbariServer 汇报本机的大数据组件运行状态，显然这种进程管理和日志查看功能使用 Python 编写最容易。

正如图 3-2 所示，AmbariServer 整合各 Agent 汇报的心跳信息，并通过 RESTAPI 向外提供 Web 服务；Agent 则用来管理本 Agent 所在机所有大数据进程。

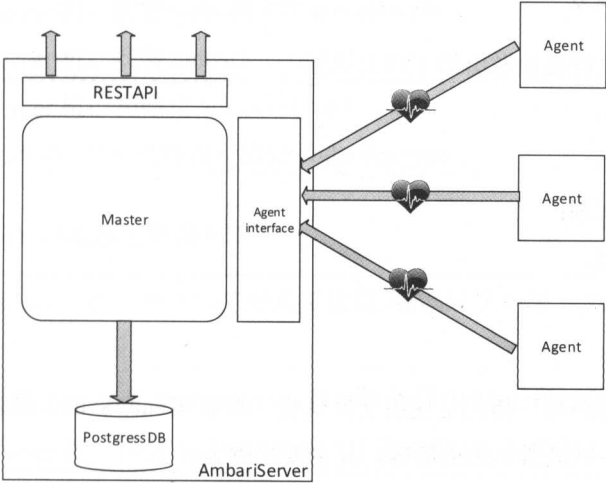


图 3-2 Ambari 体系架构

3) Ambari 部署

理论上，须在 cmaster0 上部署 AmbariServer，在 cslaveX 上部署 AmbariAgent。不过由于 AmbariAgent 是用来管理各大数据组件的，故 AmbariAgent 应当和本机大数据组件一起部署，也就是说，Ambari 的部署指的是 cmaster0 上部署 AmbariServer。

3.2 使用 Ambari 部署 HDP

本节主要讲解大数据集群搭建步骤，将讲述如何从无到有搭建一个大数据集群，不过本节主要是从理论上指导大数据集群搭建过程，不会涉及实际操作，在下一节中，将以 littleCstor 为例，实际操作整个搭建过程。

使用 Ambari 部署 HDP^[2]主要分为：制定规划→搭建硬件集群→本地建仓→部署 Ambari→使用 Ambari 部署 HDP，这五大步骤，其中在每个大步骤内又包含诸多小步骤，

除了第一大步骤“制定规划”外，其他步骤都为实际的物理操作并有相应的操作结果：

- ①制定规划；
- ②集群搭建；
- ③在 repo 机上建立本地仓库；
- ④在 AmbariServer 机上部署 ambari-server；
- ⑤使用 Ambari 在集群上部署 HDP。

1. 制定部署规划

搭建大数据集群必须做出如下四大规划：

- ①集群规划；
- ②仓库规划；
- ③Ambari 规划；
- ④HDP 规划。

2. 集群搭建

所谓的“集群搭建”指的是使用交换机将（新装 CentOS 的）硬件机器连接成局域网，并初步设置各机机器名防火墙等，其主要步骤为：

- ①针对每台机器：安装 OS。
- ②针对所有机器：通过交换机或路由器将所有机器连成局域网。
- ③针对每台机器：修改机器名，关闭防火墙，同步本机时钟，关闭 SELinux、PackageKit，验证 umask Value，关闭桌面（可选），重启本机生效上述操作。
- ④针对所有机器：添加域名映射。

3. 在 repo 上建立本地仓库

通俗地讲，“在 repo 上建立本地仓库”指的是将所有大数据组件安装包下载到一台机器上。这一步非常重要，试问没有 qq.apk，又怎么可能在 Android 上安装 QQ。而且全部大数据安装包多达 4G，必须提前下载，下面为建仓步骤：

- ①选定 repo 机。
- ②任意机：下载 Ambari、HDP、HDP-UTILS。
- ③repo 机：安装 httpd。
- ④repo 机：将 Ambari 等拷贝至 repo 机“/var/www/html”目录。
- ⑤repo 机：解压 Ambari、HDP、HDP-UTILS。

⑥针对所有机器：验证仓库 URL。

4. 在 AmbariServer 机上部署 ambari-server

既然要使用 Ambari 来自动安装大数据套件,那前提肯定是要有 Ambari,有了 Ambari 后,才有资格谈“使用 Ambari 来部署大数据套件”,本步骤的目的就是安装 Ambari,其主要可归纳为如下步骤:

- ①选定 AmbariServer 机。
- ②AmbariServer 机: 配置 yum 源文件 ambari.repo。
- ③AmbariServer 机: 安装 ambari-server。
- ④AmbariServer 机: 初始化 ambari-server。
- ⑤集群任意机 FireFox 浏览器: 验证 ambari-server。

5. 使用 Ambari 在集群上部署 HDP

在拥有了 Ambari 之后,理所当然就是要使用 Ambari 来部署大数据套件了,本步操作可归纳如下:

- ①针对所有机器: 打通 AmbariServer 机到其他机 SSH 无密钥登录。
 - ②AmbariServer 机: 拷贝本机“/root/.ssh”目录下 id_rsa 至桌面。
 - ③AmbariServer 机: 启动 ambari-server。
 - ④集群任意机 FireFox 浏览器: 打开“AmbariServer:8080”。
 - ⑤Ambari 主页: 输入用户名、密码,登录。
 - ⑥Ambari 主页: 根据向导建立大数据集群。
- Ambari 主页部署向导注意点: 务必修改默认仓库位置、上传 id_rsa。

至此,“大数据集群”部署步骤讲解结束,从上述步骤中不难看出,最后一步实际上才是真正部署 HDP,事实上的确是如此,下面给出这几个步骤之间的相互关系。

显然“集群搭建”是个独立操作,它并不依赖其他步骤,其前提是硬件机器、CentOS-6.6、路由器或交换机,结果是一个局域网集群。“在 repo 机上建立本地仓库”是个独立操作,本质上其并不依赖其他步骤,其前提是 repo 机和互联网,结果是大数据套件所有安装包都被下载到了本机。“在 AmbariServer 机上部署 ambari-server”是个独立操作,不过此步需要用到 repo 机上 Ambari 安装包,即在完成“建立本地仓库”后才能执行“在 AmbariServer 机上部署 ambari-server”。其前提是 AmbariServer 机和 repo 机上 Ambari 安装包,结果是将 Ambari 安装包安装到了 AmbariServer 机上。“使用 Ambari 在集群上部署 HDP”是个独立操作,其依赖前四个步骤,即在完成“集群搭建”、“在 repo 机上建立本地仓库”、“在 AmbariServer 机上部署 ambari-server”后,才可执行“使用 Ambari 在集

群上部署 HDP”。其前提是集群、repo 机、AmbariServer 机，结果是大数据集群。一句话描述为：AmbariServer 机上的 Ambari 工具，将 repo 机上的所有大数据安装包，安装到了集群中各台物理机器上。

本节给出了大数据集群搭建步骤，下一节中将按照此步骤实际搭建 littleCstor。

3.3 使用 Ambari 搭建 littleCstor

littleCstor 的搭建过程相当复杂，最好能分步部署，本节以 2.2 节给出的部署五大步骤为理论指导，一步步搭建 littleCstor。

3.3.1 相关约定

本节将使用 9 台裸机、1 台交换机，一步步讲述大数据集群搭建步骤。第一步将多机器连成局域网，并称此局域网集群为 prelittleCstor。最后一步，将使用 Ambari 在 prelittleCstor 上部署 HDP。待部署好 HDP 后，大数据集群也就部署成功了，编者称最终（部署好 HDP）的集群为 littleCstor。

所谓的 prelittleCstor 指的是经过如下一系列操作后的集群：在硬件裸机上安装操作系统→将安装好 OS 的机器全都连接到交换机/路由器下，设置各机机器名等。请注意 prelittleCstor 上并未安装 Ambari 或 HDP，而 littleCstor 则是在 prelittleCstor 基础上安装了 HDP。

除了名称约定，还要遵循相关软件和硬件约定，见表 3-3 和表 3-4。

表 3-3 软件约定

	约 定
Ambari 版本	决定其他组件版本，此处为：ambari-2.0.1-centos6.tar.gz
HDP 版本	依赖 Ambari 版本，此处为：HDP-2.2.6.0-centos6-rpm.tar.gz
HDP-UTILS 版本	依赖 Ambari 版本，此处为：HDP-UTILS-1.1.0.20-centos6.tar.gz
JDK 版本	依赖 Ambari 版本，此处为：jdk-7u67-linux-x64.tar.gz
OS 版本	依赖 Ambari 版本，此处为：CentOS-6.7-x86_64
CentOS 用户	root 用户、allen 用户

表 3-4 硬件约定

	约 定
集群机器数	11 台，系统统一为 CentOS-6.7-x86_64
编者自己的电脑	两台，一台为 CentOS-6.7-x86_64，另一台为 Win7
内存配置	cmaster0、cmaster1、cmaster2 为 6G
	cslave0、cslave1、cslave2、cslave3、cslave4、repo 为 4G
	iclient0 为 2G
网络-交换机	端口充足，此处为 TP-LINK 牌 16 口百兆交换机：TL-SF1016D
网络-互联网	每台机器均可以连到互联网（必选项）

此外，由于集群中机器名过长，当书中需要书写机器名时，只写出机器前缀名，表 3-5 为机器全名和缩写对应表。

表 3-5 缩写约定

缩 写	全 名
cmaster0	cmaster0.cloudlab.njupt.edu.cn
cmaster1	cmaster1.cloudlab.njupt.edu.cn
cmaster2	cmaster2.cloudlab.njupt.edu.cn
cslave0	cslave0.cloudlab.njupt.edu.cn
cslave1	cslave1.cloudlab.njupt.edu.cn
cslave2	cslave2.cloudlab.njupt.edu.cn
cslave3	cslave3.cloudlab.njupt.edu.cn
cslave4	cslave4.cloudlab.njupt.edu.cn
iclient	iclient0.cloudlab.njupt.edu.cn
repo	repo.cloudlab.njupt.edu.cn
excellent	excellent
brilliant	brilliant

有了上述约定后，下面按 3.2 节给定的五大步骤，部署大数据集群，下面为第一步“制定部署规划”。

3.3.2 制定部署规划

根据编者实验室硬件条件，编者做出如下大数据集群规划。

1. 规划 prelittleCstor

prelittleCstor 中将包含 9 台机器和 1 台交换机，交换机上层为外网路由器，这 9 台机按其命名在集群中充当不同角色。

cmaster0、cmaster1、cmaster2、cslave0、cslave1、cslave2、cslave3、cslave4、iclient0、repo

2. 仓库规划

选定 repo 机，在 repo 机上建立仓库。

3. 规划 Ambari

表 3-6 为 Ambari 规划，读者可能有疑问，既然选择 cmaster0 作为 Ambari 主节点，为何还要让此节点充当 Ambari 从节点，如果将 AmbariServer 部署到一台独立主机上，明显集群物理拓扑清晰，方便理解。此处原因很简单，编者并未购置新机器，故将 AmbariServer 部署到了 cmaster0 上。

表 3-6 Ambari 部署规划表

组件名	角 色	进 程 名	机 器
Ambari	主节点	AmbariServer	cmaster0
	从节点	AmbariAgent	cmaster0、cmaster1、cmaster2、cslave0 cslave1、cslave2、cslave3、cslave4、iclient0

4. 规划 littleCstor

littleCstor 共包括 8 台机器，这 8 个台机器分为 3 主 5 从，可根据机器名辨别出（表 3-7）。repo 不属于 littleCstor，严格来说 iclient0 也不属于 littleCstor，不过编者有时会将 iclient0 计算在内有时则不会，请读者不要在此问题上纠结。编者的 cslave2 机实际上只有 2G 内存，这直接导致 HBase 服务起不来，集群一直报警。

表 3-7 规划 littleCstor

定 位	角 色	机 器	单机配置
仓库	仓库节点	repo	4 核、4G
Ambari	主节点	cmaster0	4 核、6G
集群	主节点	cmaster0、cmaster1、cmaster2	4 核、6G
	从节点	cslave0、cslave1、cslave2、cslave3、cslave4	4 核、4G
	客户节点	iclient0	4 核、2G

至此，四步规划结束，下面即按照这四步规划搭建大数据集群。

3.3.3 搭建 prelittleCstor

搭建 prelittleCstor 的目的是将一堆硬件裸机搭建成完整局域网，并设置各机机器名，

IP 等。简单地说，本部分输入为一堆硬件裸机，输出为处于同一网络下的机器集群（且集群中各机都有自己的机器名和 IP 地址）。

(1) prelittleCstor 规划

根据部署规划，prelittleCstor 应为 9 台 PC、一台交换机，交换机上层为路由器，各机机器名前缀如下：

```
cmaster0、cmaster1、cmaster2、cslave0、cslave1、cslave2、cslave3、cslave4、iclient0、repo
```

(2) prelittleCstor 搭建步骤

- ①针对每台机器：安装 OS。
- ②针对所有机器：通过交换机将所有机器连成局域网。
- ③针对每台机器：修改机器名，关闭防火墙，同步本机时钟，关闭 SELinux、PackageKit，验证 umask Value，关闭桌面（可选），重启本机生效上述操作。
- ④针对所有机器：添加域名映射。

下面即按照 prelittleCstor 预定规划和搭建步骤，编者操作如下。

1. Step1

- ①针对每台机器：安装 OS。

请读者自行完成此步，请务必确保 OS 版本为 CentOS-6.7-x86_64，系统语言选定 English。安装时用户名为 allen，密码为 allen，此时 root 会自身设置自身密码为安装用户 allen 的密码 allen。

2. Step2

- ②针对所有机器：通过交换机将所有机器连成局域网。

此步看似简单，实际上极为复杂，没有扎实的计算机网络和 Linux 知识，最终将无法搭建局域网。对于仅有一个网卡的单机，可根据路由器不同配置，分为静态网络和动态网络两种情况配置单机网络，对于多网卡的服务器，网络配置则更加复杂。本书只讲解单网卡设置（图 3-3），对于多网卡设置，请参看专业书籍。

在计算机网络中，对于网络上的单机来说，交换机是透明的，故本机 IP 配置只和距离本机最近的路由器有关，下面根据路由器不同设置，配置单机 IP 地址。

1) 路由器开启 DHCP 服务

路由器开启 DHCP 服务时，单机会自动从路由器获取动态 IP 地址，不过新装的 CentOS 默认不开启网络服务，须进行下述配置，设置本机开机后自动启动网卡 eth0，设置后请务必重启本机网络使之生效。

```
#九台机器都须执行
```

```
[root@localhost ~]# vim /etc/sysconfig/network-scripts/ifcfg-eth0 #编辑本机网卡 eth0, root 权限
```

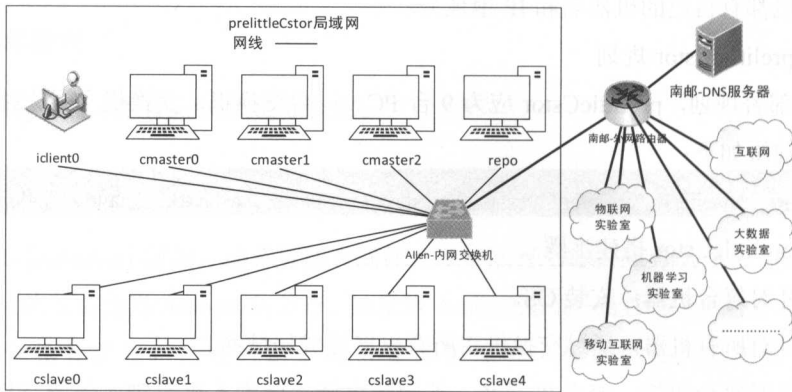


图 3-3 prelittleCstor 网络拓扑图

将内容里的“ONBOOT=no”替换为“ONBOOT=yes”，接着重启 eth0。

```
#九台机器都须执行
[root@localhost ~]# service network restart #启动网卡 eth0, root 权限
..... #省略输出
[root@localhost ~]# ifconfig #查看本机 IP 地址, 任意权限, 下面为命令输出
eth0      Link encap:Ethernet  HWaddr 40:61:86:7F:FE:EB
          inet addr:192.168.1.112  Bcast:192.168.1.255  Mask:255.255.255.0
          .....
```

eth0 重启后，可通过命令“ifconfig”验证是否配置成功，从输出中我们看到本机已分配到动态 IP 地址“192.168.1.112”，至此单机 IP 配置结束，如果读者上层路由器开启了 DHCP，请参考上述步骤，对集群内九台机器，设置单机网络。

2) 路由器未开启 DHCP 服务

路由器未开启 DHCP 时，其下单机不会自动获取到 IP 地址。此时，应根据路由器规定的 IP 地址段分别为各单机设置静态 IP。编者所在机构南京邮电大学的路由器就为此类型，且单机 IP 地址约定如下：

```
单机 IP 地址段：10.10.201.1~10.10.201.255
路由器内网 IP 地址：10.10.201.20
子网掩码：255.255.255.0
DNS：202.119.230.8
```

故编者使用上述地址段，按下述步骤，对集群中的九台机器，每台都设置了一个静态 IP 地址。

One 编辑网卡

```
#九台机器都须执行
[root@localhost ~]# vim /etc/sysconfig/network-scripts/ifcfg-eth0 #编辑本机网卡 eth0, root
```


权限

将原内容修改为下述内容:

```
DEVICE=eth0
TYPE=Ethernet
UUID=勿修改
ONBOOT=yes
NM_CONTROLLED=yes
BOOTPROTO=static
HWADDR=勿修改
IPADDR=10.10.201.100
PREFIX=24
GATEWAY=10.10.201.20
NETMASK=255.255.255.0
DNS1=202.119.230.8
DEFROUTE=yes
IPV4_FAILURE_FATAL=yes
IPV6INIT=no
NAME="System eth0"
```

请务必注意不要修改本机网卡硬件地址 HWADDR 和 UUID。对于这九台机器,除了 HWADDR 和 UUID 不同外,“IPADDR=10.10.201.106”字段也需要不断更改,例如使用 100~112 这 13 个 IP (9 个集群 IP, 另外 4 个其他用途), 设置完本机静态 IP 地址后,同样需要重启本机网卡。

Two 重启本机网卡

```
#九台机器都须执行
[root@localhost ~]# service network restart      #启动网卡 eth0, root 权限
.....                                           #省略输出
[root@localhost ~]# ifconfig                    #查看本机 IP 地址, 任意权限, 下面为命令输出
eth0      Link encap:Ethernet  HWaddr D4:3D:7E:D1:0D:81
           inet addr:10.10.201.100  Bcast:10.10.201.255  Mask:255.255.255.0
           .....

```

Three 验证是否配置成功

eth0 重启后, 可通过命令 “ifconfig” 验证是否配置成功, 从输出中我们看到已成功设置本机静态 IP 地址 “10.10.201.100”, 至此单机静态 IP 配置结束, 如果读者上层路由未开启 DHCP, 请读者对集群内这九台机器, 每台都执行上述设置, 为各机配置静态 IP 地址。

经过配置后, 九台机器 IP 地址分别为:

```
10.10.201.100; 10.10.201.101; 10.10.201.102
10.10.201.103; 10.10.201.104; 10.10.201.105; 10.10.201.107; 10.10.201.109
10.10.201.106; 10.10.201.115
```


3. Step3

先以 IP 地址为“10.10.201.100”的机器为操作对象,将此机的机器名修改为 `cmaster0`,接着,在此机器上执行其他操作,这些操作合计为:

③修改机器名,关闭防火墙,同步本机时钟,关闭 SELinux、PackageKit,验证 `umask Value`,关闭桌面(可选),重启本机生效。

1) 修改机器名

修改 CentOS 机器名时须更改 `network` 文件,修改后重启生效。具体操作为,在 IP 为“10.10.201.100”的机器上,打开命令行,以 `root` 身份执行如下操作:

```
[root@localhost ~]# vim /etc/sysconfig/network #编辑机器名文件,root 权限
```

找到“`HOSTNAME`”这一行,将装机时默认机器名“`localhost.localdomain`”修改成“`cmaster0.cloudlab.njupt.edu.cn`”,请注意应全部使用小写字母,且切勿修改或删除其他内容,修改后“`HOSTNAME`”一行对应内容如下:

```
HOSTNAME=cmaster0.cloudlab.njupt.edu.cn
```

此操作重启后才能生效,不过读者可暂不重启机器,待部署结束时编者将提示重启。

2) 关闭防火墙

永久关闭 CentOS 防火墙须用到 `chkconfig` 命令,此命令重启后生效。具体操作为,在 IP 为“10.10.201.100”的机器上,进入命令行,以 `root` 身份执行如下操作:

```
[root@localhost ~]# chkconfig iptables off #永久关闭防火墙,root 权限
```

此操作重启后才能生效,部署结束时编者会提示重启。

3) 同步本机时钟

CentOS 使用 `ntpd` 服务同步本机时钟,操作时在“10.10.201.100”机器上,执行下述命令,在执行命令时,请务必保证本机能够连接互联网:

```
[root@localhost ~]# yum install ntpd #安装 ntpd,root 权限
```

```
[root@localhost ~]# service ntpd start #启动 ntpd,root 权限
```

```
[root@localhost ~]# chkconfig ntpd on #设置开机自动启动 ntpd,root 权限
```

此操作无须重启,自动生效,不过读者要注意,此服务是 `ntpd`,而不是 `ntpdate`。

4) 关闭 SELinux

默认情况下,CentOS 自动开启 SELinux 服务,下述命令通过修改 SELinux 配置文件来关闭 SELinux,此命令重启后生效。操作时,在“10.10.201.100”上执行:

```
[root@localhost ~]# vim /etc/selinux/config #编辑 selinux 配置文件,root 权限
```

找到“`SELINUX=enforcing`”并替换如下:

```
SELINUX=disabled
```

切勿修改其他内容，此操作重启后生效，稍后编者会提示。

5) 关闭 PackageKit

在“10.10.201.100”机上，进入命令行，编辑 PackageKit 配置文档，做如下修改：

```
[root@localhost ~]# vim /etc/yum/pluginconf.d/refresh-packagekit.conf #root 权限
```

将“enabled=1”，替换为：

```
enabled=0
```

6) 验证 umask Value

umask (user file-creation mode mask) 值用来设置新建文件时系统默认赋予的文件访问权限，大部分 Linux 发行版都设置为“022”，CentOS 在此处有所不同，必须修改，编辑“10.10.201.100”上 profile 文件：

```
[root@localhost ~]# vim /etc/profile #编辑 profile 文件,root 权限
```

在文件末尾追加如下内容，其他内容切勿修改：

```
umask 022
```

7) 关闭桌面 (可选)

```
[root@localhost ~]# vim /etc/inittab #修改默认启动级别,root 权限
```

以 root 权限修改启动级别文件 inittab，找到“id:5:initdefault:”并将“5”改成“3”，即：

```
id:3:initdefault:
```

此操作为可选操作，可以不做，但桌面一般要占用近 512M 内存，如果不启动桌面，能大大降低系统内存使用量，和上述操作类似，此操作也需要在重启后方能生效。

8) 重启生效

当执行完上述操作后，读者需要重启 IP 地址为“10.10.201.100”的机器，确保上述操作全部生效。

```
[root@localhost ~]# reboot #重启机器,生效所有操作,root 权限
```

如果读者执行了“关闭桌面”操作，那么重启后，默认将直接进入命令行模式，此时读者可执行如下命令进入窗体界面：

```
[root@cmaster0 ~]# startx #手动开启桌面
```

重启后，由于刚才的操作全部生效，打开命令行后，命令行会改为：

```
[root@cmaster0 ~]#
```

读者原来看到的“localhost”已经改成“cmaster0”，由于命令行会自动省略第一逗号后面的“cloudlab.njupt.edu.cn”，所以只显示了“cmaster0”。读者可通过如下命令逐一验证上述操作：

```
[root@cmaster0 ~]# hostname #验证机器名
[root@cmaster0 ~]# service iptables status #验证防火墙,root 权限
[root@cmaster0 ~]# service ntpd status #验证时钟同步服务,root 权限
[root@cmaster0 ~]# sestatus -v #验证 selinux,root 权限
```

至此，已完成单机“10.10.201.100”上的步骤③操作，请读者务必保证上述操作都已执行并正确执行，否则部署 HDP 时将会报错。

按要求，prelittlCstor 中的九台机器都要执行步骤③，由于当前仅对机器“10.10.201.100”进行了上述操作，故还要对余下（表 3-8）的八台机器，逐一执行上述操作。

表 3-8 机器名与 IP 地址对应表

IP 地址	机器名
10.10.201.101	cmaster1.cloudlab.njupt.edu.cn
10.10.201.102	cmaster2.cloudlab.njupt.edu.cn
10.10.201.103	cslave0.cloudlab.njupt.edu.cn
10.10.201.104	cslave1.cloudlab.njupt.edu.cn
10.10.201.105	cslave2.cloudlab.njupt.edu.cn
10.10.201.107	cslave3.cloudlab.njupt.edu.cn
10.10.201.109	cslave4.cloudlab.njupt.edu.cn
10.10.201.106	iclient0.cloudlab.njupt.edu.cn
10.10.201.115	repo.cloudlab.njupt.edu.cn

请读者参照“10.10.201.100”机上的操作步骤，对上述八台机器，执行步骤③，即：

③针对余下的八台机器：修改机器名，关闭防火墙，同步本机时钟，关闭 SELinux、PackageKit，验证 umask Value，关闭桌面（可选），重启本机生效上述操作。

至此，prelittlCstor 的步骤③操作结束。此时 prelittlCstor 集群中每台机器都有了己的 IP 地址、机器名，且各机均处于同一网段。

4. Step4

④针对所有机器：添加域名映射。

互联网上不同机器之间相互通信时需要用到机器名，以 cmaster0 为例，当 cmaster0 上进程和 cslave0 上进程通信时，cmaster0 会将请求组织成类似如下格式并发送出去：

cslave0.cloudlab.njupt.edu.cn:端口号:协议:数据

可是在计算机网络中，应用层以下只识别 IP 地址，如果不能将此机器名 cslave0.cloudlab.njupt.edu.cn 转换成 IP 地址 10.10.201.103，此数据根本无法从本机发出。当前主要有“DNS 自动查询”和“本机手工配置”两种方式实现<IP 地址、机器名>转换。

互联网上的众多 DNS 服务器即负责机器名到 IP 的转换工作，内部 DNS 服务器维护着一张<机器名、IP 地址>映射表，通过此表即可完成映射。下面根据用户“有无权限配置本机构的 DNS 服务器”，讲述添加域名映射。

1) 有权限配置 DNS 映射表

此时只需要在本机构 DNS 服务器上添加<IP、机器名>，不需要对 prelittleCstor 的 9 台机器进行任何配置。具体操作时，读者须将下面的映射文件改成 DNS 识别的 zone 嵌套格式并加入 DNS 配置文件。

```
10.10.201.100    cmaster0.cloudlab.njupt.edu.cn
10.10.201.101    cmaster1.cloudlab.njupt.edu.cn
10.10.201.102    cmaster2.cloudlab.njupt.edu.cn

10.10.201.103    cslave0.cloudlab.njupt.edu.cn
10.10.201.104    cslave1.cloudlab.njupt.edu.cn
10.10.201.105    cslave2.cloudlab.njupt.edu.cn
10.10.201.107    cslave3.cloudlab.njupt.edu.cn
10.10.201.109    cslave4.cloudlab.njupt.edu.cn

10.10.201.106    iclient0.cloudlab.njupt.edu.cn
10.10.201.115    repo.cloudlab.njupt.edu.cn
```

当 cmaster0 向 cslave0 发送数据时，cmaster0 会先向 DNS 服务器询问 cslave0 机真实的 IP 是什么。在获得 cslave0 的 IP 后，cmaster0 将请求头的机器名替换成 IP 地址并将 IP 数据报发往上层路由器，转换过程如下：

```
cslave0.cloudlab.njupt.edu.cn:端口号:协议:数据
DNS...
10.10.201.103:端口号:协议:数据
```

2) 无权限配置 DNS 映射表

此时只能配置单机的域名映射文件“/etc/hosts”，且须对 prelittleCstor 内的九台机器都进行配置，下面以 cmaster0 为例，实现在 cmaster0 上添加域名映射：

```
[root@cmaster0 ~]# vim /etc/hosts                                #编辑域名映射文件,root 权限

编辑 hosts 文件，将如下内容追加到 hosts 文件末尾，注意是追加，勿修改其他内容：
10.10.201.100    cmaster0.cloudlab.njupt.edu.cn
10.10.201.101    cmaster1.cloudlab.njupt.edu.cn
10.10.201.102    cmaster2.cloudlab.njupt.edu.cn

10.10.201.103    cslave0.cloudlab.njupt.edu.cn
10.10.201.104    cslave1.cloudlab.njupt.edu.cn
10.10.201.105    cslave2.cloudlab.njupt.edu.cn
10.10.201.107    cslave3.cloudlab.njupt.edu.cn
10.10.201.109    cslave4.cloudlab.njupt.edu.cn
```

```
10.10.201.106    iclient0.cloudlab.njupt.edu.cn
10.10.201.115    repo.cloudlab.njupt.edu.cn
```

当 `cmaster0` 以机器名方式访问 `cslave0` 时, `cmaster0` 会查找本机上的 `hosts` 文件并根据文件内容将机器名替换为 IP 地址。编者并无权限修改南邮内网域名映射服务器, 故 `prelittleCstor` 集群中采用该方式配置域名映射。

至此, 已成功在 `cmaster0` 上添加了所有机器的域名映射, 下一步请读者自行在另外八台机上执行此操作, 即对如下机器, 将上述文件追加到各自 `“/etc/hosts”` 文件末尾。

```
cmaster1、cmaster2、cslave0、cslave1、cslave2、cslave3、cslave4、iclient0、repo
```

当 9 台机器都执行完步骤④后, `prelittleCstor` 集群的步骤④配置宣告结束, 此时整个 `prelittleCstor` 搭建结束。

本节通过四个 Step, 成功地将九台硬件机器搭建成了 `prelittleCstor`。简单地说, 本步骤的输入是一堆硬件裸机, 输出是局域网集群。下一节, 将使用这九台机器里的 `repo` 机, 实现在此机上建立本地仓库。

3.3.4 本地建仓

本地建仓的目的是将所有大数据组件安装包, 下载到本地某台机器上 (这里选择下载到 `repo` 机器上)。这样, 当集群中某台机器需要下载大数据组件安装包时, 就可以直接到 `repo` 机器上下载, 而无须到互联网上下载。

本步骤为必选项, 完整的大数据套件安装包多达 4G, 就算是使用迅雷都要下载近 2 小时, 如果不事先下载好, 等到部署时到互联网上下载, 会发生下载超时出错。

简单地说, 本部分输入为 `repo` 机和互联网上的 HDP 安装包, 输出为将这些大数据安装包下载到 `repo` 机。

(1) “本地建仓”的具体规划

编者的 `prelittleCstor` 集群中有台实体机器 `repo` 机, 选用此机器建立本地仓库, 故“本地建仓”规划为:

- 在 `repo` 机上建立本地仓库, 仓库中存在 Ambari、HDP、HDP-UTILS。
- 其中 Ambari 版本为 `ambari-2.0.1`, HDP 和 HDP-UTILS 版本根据 Ambari 版本而定。
- (2) 根据此规划, 下面给出建立本地仓库主要步骤
- ①选定 `repo` 机。
- ②任意机: 下载 Ambari、HDP、HDP-UTILS。
- ③`repo` 机: 安装 `httpd`。

④repo 机：将 Ambari 等拷贝至 repo 机的“/var/www/html”目录。

⑤repo 机：解压 Ambari、HDP、HDP-UTILS。

⑥集群机：验证仓库 URL。

下面根据此规划和建仓步骤，具体讲述建仓过程。

1. Step1

①选定 repo 机。

此步骤很简单，选定 repo 机后，请确保此机能够访问互联网。和其他单机不同，repo 机最稀缺的资源是带宽。建仓时 repo 机要下载近 4G 大数据套件安装包，部署 HDP 时集群时其他机器都到 repo 机上下载该安装包文件，故其对带宽需求较高。编者无权配置路由器带宽，故将 repo 内存设置在 4G 左右，尽量加快数据读写。

2. Step2

②任意机：下载 Ambari、HDP、HDP-UTILS。

下面三条命令为在 repo 机上使用 wget 命令下载上述三个压缩包。输入第一条命令后，回车，wget 会下载 ambari-2.0.1-centos6.tar.gz，此文件大小大概为 230M。由于此命令不会提示下载进度，可能会“卡住”半小时，读者可能以为命令假死，实际上正在下载。第二条命令下载 HDP-2.2.6.0-centos6-rpm.tar.gz，此文件大小近 4G，回车后，可能会“卡住”近 6 小时。第三条命令下载 HDP-UTILS-1.1.0.20-centos6.tar.gz，此文件较小，五分钟内即可完成。

```
[root@repo ~]# wget -nv http://public-repo-1.hortonworks.com/ambari/centos6/2.x/updates/2.0.1/ambari-2.0.1-centos6.tar.gz
[root@repo ~]# wget -nv http://public-repo-1.hortonworks.com/HDP/centos6/HDP-2.2.6.0-centos6-rpm.tar.gz
[root@repo ~]# wget -nv http://public-repo-1.hortonworks.com/HDP-UTILS-1.1.0.20/repos/centos6/HDP-UTILS-1.1.0.20-centos6.tar.gz
```

这三条命令都在 repo 机上以 root 身份执行，由于文件太大，加上网络带宽受限，整个下载过程会持续近 6 小时。编者此处做法较为“聪明”，具体操作是，先在笔记本电脑上（brilliant 机），用迅雷下载这三个文件，然后用 U 盘将这三个文件 copy 至 repo 机。Win7 下的迅雷下载这三个文件大概需要 2 小时，大大缩短了下载时间。既然操作的目的是将三个压缩文件下载至 repo 机，那么到底选用 wget 还是迅雷，都无关要紧，只要确保下载成功并将文件存放到 repo 机上即可。

3. Step3

③repo 机：安装 httpd。

安装之前,读者可打开 repo 机上的 FireFox 浏览器,地址栏输入“localhost”,FireFox 会提示出错,说明本机并未安装 httpd,不过即使安装了也可以再次安装。下面使用 yum 命令在 repo 机上安装 httpd 服务,执行命令时请务必确保 repo 可正常连接互联网。

```
[root@repo ~]# yum install httpd                #安装 httpd,repo 机,root 权限
[root@repo ~]# service httpd start              #启动 httpd,repo 机,root 权限
[root@repo ~]# chkconfig httpd on                #设置开机自动启动 httpd
```

在 repo 机上执行上述三条命令后,httpd 安装结束。此时打开 repo 机上 FireFox 浏览器输入“localhost”回车后,会显示 Apache 测试页,此页证明 httpd 安装并开启成功。

4. Step4

④repo 机: 将 Ambari 等拷贝至 repo 机的“/var/www/html”目录。

假定下载好的三个压缩文件在“/root”目录下,读者须将 ambari 拷贝至 html 目录下,然后新建“html/hdp”目录并将 HDP 和 HDP-UTILS 拷贝至此目录下,具体操作如下:

```
[root@repo ~]# cd /var/www/html/                #进入/var/www/html 目录
[root@repo html]# mv /root/ambari-2.0.1-centos6.tar.gz ./ #移动 ambari 至 html 目录
[root@repo html]# mkdir hdp                      #新建 hdp 目录
[root@repo html]# cd hdp/                        #进入 hdp 目录
[root@repo hdp]# mv /root/HDP-2.2.6.0-centos6-rpm.tar.gz ./ #移动 HDP 至 hdp 目录
[root@repo hdp]# mv /root/HDP-UTILS-1.1.0.20-centos6.tar.gz ./ #移动 HDP-UTILS 至 hdp 目录
```

上述命令须在 repo 机上以 root 用户执行,命令主要完成将 ambari 移至 html 目录,HDP 和 HDP-UTILS 移至 hdp 目录。命令结束后,“/var/www/html”下存在目录“ambari-2.0.1”和“hdp”,而“hdp”目录下出现文件夹“HDP”和“HDP-UTILS-1.1.0.20”。

5. Step5

⑤repo 机: 解压 Ambari、HDP、HDP-UTILS。

此步骤须将 ambari 解压至 html 目录下,另外两个文件解压到 hdp 目录,操作如下:

```
[root@repo ~]# cd /var/www/html/                #进入/var/www/html 目录
[root@repo html]# tar -zxvf ambari-2.0.1-centos6.tar.gz #将 ambari 文件解压至当前目录
[root@repo html]# mv ambari-2.0.1-centos6.tar.gz /root/ #移出 ambari 文件
[root@repo html]# cd hdp/                        #进入 hdp 目录
[root@repo hdp]# tar -zxvf HDP-2.2.6.0-centos6-rpm.tar.gz #解压 HDP 至当前目录(hdp)
[root@repo hdp]# tar -zxvf HDP-UTILS-1.1.0.20-centos6.tar.gz #解压 HDP-UTILS 至当前目录(hdp)
[root@repo hdp]# mv HDP-2.2.6.0-centos6-rpm.tar.gz /root/ #移出 HDP 文件
[root@repo hdp]# mv HDP-UTILS-1.1.0.20-centos6.tar.gz /root/ #移出 HDP-UTILS 文件
```

上述命令须在 repo 机上以 root 用户执行,命令主要完成将 ambari 解压至 html 目录,将 HDP 和 HDP-UTILS 解压至 hdp 目录。

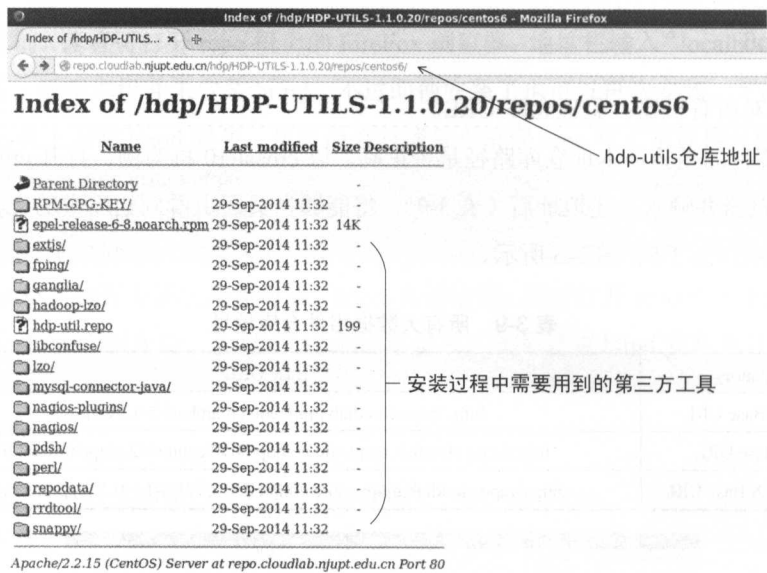


图 3-6 hdp-utils 本地仓库 URL 效果图

• 验证“Ambari Base URL”地址：确保集群中所有机器都能够访问该地址。这里的所有指的是 cmasterX, cslaveX, iclientX, brilliant 和 excellent 机。其实，只要某机和 repo 机位于同一局域网，那么该机就一定能够访问上述地址。

- 验证“HDP Base URL”地址：确保集群中所有机器都能够访问该地址。
- 验证“HDP-UTILS Base URL”地址：确保集群中所有机器都能够访问该地址。

按要求，集群中九台机器都要检验此 URL 地址，请读者对下面的机器，打开对应机 Firefox 浏览器，地址栏输入上述几个 URL 地址，进行验证。

repo、cmaster1、cmaster2、cslave0、cslave1、cslave2、cslave3、cslave4、iclient0

实际上 repo 机器应当最先验证，因为当 repo 本机本身都不成功时，其他机器更不可能成功。当集群中九台机器都已验证了仓库 URL，且都可打开时，本地建仓成功。

本部分通过六个 Step 讲述了 repo 机上建立本地仓库的完整过程，仓库中存放了 Ambari 和 HDP 两类安装包，接下来讲述的“安装 AmbariServer”和“部署 HDP”中用的 Ambari 安装包和 HDP 安装包就是从 repo 机上获取的。

3.3.5 部署 AmbariServer

第一节已说明 Ambari 采用 master/slave 架构，主节点运行 master 进程 AmbariServer，从节点运行 slave 进程 AmbariAgent。不过，Ambari 部署时却只需要先在主节点上部署 AmbariServer，AmbariAgent 在页面“部署向导”里操作完成。这是因为 AmbariServer 和

AmbariAgent 服务对象不同。AmbariServer 采用 Java 编写，面向用户（人），AmbariAgent 由 Python 编写，核心思想是用脚本管理（启动、查看、关闭）大数据进程和监控（实际上就是读写）进程输出日志。既然 AmbariAgent 要管理本机所有大数据组件，那么 AmbariAgent 就应当和本机大数据组件一起部署，这样能够大大降低部署复杂性。故所谓的部署 Ambari 就是指在主节点上部署 AmbariServer。

部署 AmbariServer 的目的是拥有 Ambari，即将 AmbariServer 安装到主节点上。这样，当需要使用 Ambari 安装 HDP 时，就可以直接使用该 Ambari。试问，如果需要用卡车拉货，现在连卡车都没有，该怎么办，Ambari 就相当于这里的卡车，部署 Ambari 就相当于创造了（或叫来了）一辆卡车。拥有 Ambari 后，才有资格谈论使用 Ambari。

简单地说，本部分输入为 repo 机 Ambari 安装包，输出为将该包安装到了主节点上。为确保集群拓扑尽量完整，应将 AmbariServer 单独部署到一台机器上，编者暂未购置新机器，故将 AmbariServer 部署到 cmaster0 上，请读者不要误解。

(1) “AmbariServer 部署”规划

为了确保集群物理拓扑尽量清晰，应当像 repo 机一样，单独购置一台机器来部署 AmbariServer，见表 3-10。

表 3-10 littleCstor 理想物理拓扑

定 位	角 色	机 器
repo	仓库节点	repo 机
Ambari	主节点	AbiMaster 机
	从节点	prelittleCstor 中除了 repo 外的九台机器
HDP	主节点	cmaster0、cmaster1、cmaster2
	从节点	cslave0、cslave1、cslave2、cslave3、cslave4
client	客户节点	iclient0、iclient1

编者在部署时并未将 AmbariServer 部署在单机上，而是部署到了 cmaster0 上，这主要是由于暂未购置新机器。此时 cmaster0 机既部署了 Ambari 主服务，又部署了 Ambari 从服务。故对于 littleCstor 来说，即将表 3-11 中的拓扑改成表 3-12 中的拓扑。

表 3-11 Ambari 完美拓扑

Ambari 主节点	独立 abimaster 机
Ambari 从节点	cmaster0, cmaster1, cmaster2, cslave0, cslave1, cslave2, cslave3, cslave4, iclient0~1

表 3-12 littleCstor 中实际 Ambari 部署拓扑

Ambari 主节点	cmaster0 机
Ambari 从节点	cmaster0, cmaster1, cmaster2, cslave0, cslave1, cslave2, cslave3, cslave4, iclient0

由于 cmaster0 机既充当 Ambari 主节点，又同时作为 Ambari 从节点，还要部署大量

HDP 主服务, 为此造成集群单机角色混乱, 给读者带来理解不便, 请谅解。如果读者机器充足, 应做出表 3-11 中的 Ambari 部署规划。不过根据编者实际情况, 只能做出表 3-12 中的 Ambari 部署规划, 即编者 Ambari 部署规划为:

在 cmaster0 上安装 AmbariServer。

在 prelittleCstor 集群中除了 repo 外的所有机器上(包括 cmaster0)部署 AmbariAgent。

(2) AmbariServer 部署步骤

前文已经说明, 实际部署 Ambari 时只部署 AmbariServer, 即在 cmaster0 上部署 AmbariServer, 而 AmbariAgent 需要和 HDP 一起部署。下面为在 cmaster0 上部署 AmbariServer 的部署步骤:

①选定 cmaster0 机。

②cmaster0 机: 配置 yum 源文件 ambari.repo。

③cmaster0 机: 安装 ambari-server。

④cmaster0 机: 初始化 ambari-server。

⑤集群任意机 FireFox 浏览器: 验证 ambri-server。

1. Step1

①选定 cmaster0 机。

此处, 编者将 AmbariServer 部署到了 cmaster0, 有条件的读者应尽量在独立机器上部署 AmbariServer。

2. Step2

②cmaster0 机: 配置 yum 源文件 ambari.repo。

当 cmaster0 需要安装 AmbariServer 时, 首先需要下载 Ambari, 既然在 prelittleCstor 中有台单机 repo 充当 Ambari、HDP 仓库, 那如何指定 cmaster0 机在下载 Ambari 安装包时直接到 repo 机下载, 而不是到 Hortonworks 官方仓库下载呢。此时就需要配置 yum 源, 通过为 yum 添加新的仓库源, 即可指定 yum 安装软件时, 到用户指定的目标 URL 下载。实际操作时, 步骤如下。

1) 下载 ambari.repo

<http://public-repo-1.hortonworks.com/ambari/centos6/2.x/updates/2.0.1/ambari.repo>

上述 URL 为 Hortonworks 官方仓库里 ambari.repo 模板文件, 请读者下载此文件。读者可通过在 cmaster0 机上执行 wget 命令、FireFox 浏览器下载, 甚至是笔记本电脑里的迅雷下载。下载好 ambari.repo 模板后, 使用编辑器打开此文件, 文件内容如下:

[Updates-ambari-2.0.1]

```

name=ambari-2.0.1 - Updates
baseurl=http://public-repo-1.hortonworks.com/ambari/centos6/2.x/updates/2.0.1
gpgcheck=1
gpgkey=http://public-repo-1.hortonworks.com/ambari/centos6/RPM-GPG-KEY/RPM-GPG-KEY-Jenkins
enabled=1
priority=1

```

第一行为容器名，第二行进一步解释容器意义，第三行的 `baseurl` 非常重要，此处指明 `yum` 命令到 Hortonworks 官方仓库下载 `ambari` 压缩包，编辑目的就是将此 URL 换成 `repo` 上的 URL。第四行指定查阅 RPM 文件内的数据证书，第五行则指定数字证书的公钥文件所在位置，第六行说明启用此容器。

2) 修改 baseurl

我们只要将第三行的 `baseurl` 值替换成本地 `ambari` 仓库 URL，其他切勿修改，替换后内容如下：

```

[Updates-ambari-2.0.1]
name=ambari-2.0.1 - Updates
baseurl=http://repo.cloudlab.njupt.edu.cn/ambari-2.0.1/centos6
gpgcheck=1
gpgkey=http://public-repo-1.hortonworks.com/ambari/centos6/RPM-GPG-KEY/RPM-GPG-KEY-Jenkins
enabled=1
priority=1

```

其中 “`baseurl=http://repo.cloudlab.njupt.edu.cn/ambari-2.0.1/centos6`” 一行，即指定当使用 `yum` 命令安装 `ambari` 时到 “`repo.cloudlab.njupt.edu.cn`” 机（即 `repo` 机）上的仓库下载 `ambari` 安装包。读者注意，`gpgkey` 的 URL 地址并不需要修改，`yum` 安装命令会下载此 URL 对应的文件进行验证，此文件非常小。HDP 部署过程中还要陆续下载一些小文件，故务必确保 `prelittlCstor` 内各机均可连接互联网。

3) 保存 ambari.repo 至特定目录

接着，保存此文件并再次确认文件名为 `ambari.repo`。下一步需要将 `ambari.repo` 拷贝至 `cmaster0` 机的 “`/etc/yum.repos.d`” 目录下，假定文件已存储在 `cmaster0` 的 “`/root`” 目录下，下述为使用 `cp` 命令将 `ambari.repo` 复制至目录 “`/etc/yum.repos.d`” 下：

```

[root@cmaster0 ~]# cp /root/ambari.repo /etc/yum.repos.d/ #复制 ambari.repo 至指定目录,root 权限

```

完成上述操作后，当使用 `yum` 命令安装 `ambari` 时，`yum` 会自动到此目录读取 `ambari.repo` 并根据文件里的 `baseurl` 地址访问 `repo` 机上的仓库（而不是远隔重洋的美国）。

3. Step3

③cmaster0 机: 安装 ambari-server。

使用 yum 安装 ambari-server 只需一条命令, 非常简单。不过在命令执行之前请读者务必确保文件 ambari.repo (已修改过 bashurl) 已经存放在 cmaster0 机的“/etc/yum.repos.d”目录下, 此外, 还要保证 cmaster0 已连接互联网, 下面为安装命令:

```
[root@cmaster0 ~]# yum install ambari-server #在 cmaster0 上安装 ambri-server,root 权限
```

当提示是否确定后, 输入“y”, 输出过程非常长, 这里不再显示, 命令输出末尾, 若看到类似如下提示, 则说明 ambari-server 在 cmaster0 上安装成功。

```
Installing : postgresql-libs-8.4.20-1.el6_5.x86_64 1/4
Installing : postgresql-8.4.20-1.el6_5.x86_64 2/4
Installing : postgresql-server-8.4.20-1.el6_5.x86_64 3/4
Installing : ambari-server-2.0.1-147.noarch 4/4
Verifying : postgresql-server-8.4.20-1.el6_5.x86_64 1/4
Verifying : postgresql-libs-8.4.20-1.el6_5.x86_64 2/4
Verifying : ambari-server-2.0.1-147.noarch 3/4
Verifying : postgresql-8.4.20-1.el6_5.x86_64 4/4
Installed : ambari-server.noarch 0:1.7.0-135
Dependency Installed:
postgresql.x86_64 0:8.4.20-1.el6_5
postgresql-libs.x86_64 0:8.4.20-1.el6_5
postgresql-server.x86_64 0:8.4.20-1.el6_5
Complete!
```

正如编者所述, 只需这一条命令即可完成安装, 虽然命令很精简, 可它依赖前期所有操作, 前面的任何一个瑕疵都可能导致本步不成功, 请读者务必确保每一步都被正确执行。

此时, cmaster0 已成功安装了 ambari-server, 不过在启动此服务前, 还要对它进行最基本设置。

4. Step4

④cmaster0 机: 初始化 ambari-server。

Ambari-server 的基本设置包括检测 SELinux 和 iptables、个性化用户名、确认 JDK 和个性化数据库等, 下述命令为初始化 ambari-server:

```
[root@cmaster0 ~]# ambari-server setup
```

回车后, 无特别需求, 请读者选择默认设置。在搭建 prelittleCstor 时, 若读者未成功关闭 SELinux, 则此处会提示 waring, 选择默认值“y”, 继续下一步。此步询问是否重新设置 AmbriServer 的所有权用户, 请保持默认设置, 继续下一步。若读者未能成功关闭

iptables, 此时又会提示 waring, 输入“y”, 继续下一步。此步非常重要, 请读者选择默认 JDK 版本, 尽量不要使用其他版本 JDK, 继续下一步。此步设置数据库, 请选择默认设置, 至此, ambari-server 初始化成功。

读者可能会有疑问, 为何 Ambari 不自动集成 JDK, 而要手动下载呢, 这是由于 JDK 和 HDP 版权不同, 为避免法律纠纷, Ambari 选择提示用户手动下载。总之 ambari-server 初始化并无特别注意点, 正常选择默认设置即可。

在 “ambari-server setup” 命令执行过程中, 其会自动到 Oracle 仓库下载 jdk-7u67-linux-x64.tar.gz, 该包近为 135M, 加之 Oracle 仓库远在美国, 极有可能因网速太慢而下载失败, 接着导致整个 setup 失败。此时, 读者可采取以下方法。

1) 手工下载 JDK

读者可使用个人电脑上的迅雷, 手工下载 jdk-7u67-linux-x64.tar.gz。

2) 将 JDK 复制至 cmaster0 机特定目录

将 PC 上刚下载的 jdk-7u67-linux-x64.tar.gz 拷贝至 cmaster0 机的 “/var/lib/ambari-server/resources/” 目录。

3) 再次执行 Step4

④cmaster0 机: 初始化 ambari-server。

也就是再次执行下述命令:

```
[root@cmaster0 ~]# ambari-server setup
```

此时脚本会检测到 JDK 已经下载好, 不会再到 Oracle 仓库去下载 JDK, 也就不会发生下载超时错误了。

5. Step5

⑤集群任意机 FireFox 浏览器: 验证 ambari-server。

在初始化 ambari-server 后, 即可使用如下命令启动 ambari-server 服务:

```
[root@cmaster0 ~]# service ambari-server start #启动 ambari-server 服务,root 权限
```

在 ambari-server 安装过程中, 安装脚本会自动将 ambari-server 设置为开机自动启动, 故用户并不需要手动添加开机启动项。那么在开启 cmaster0 时, 既然 ambari-server 已经能够自动启动了, 至于此处为何还需要手工启动, 这是因为到现在编者并未重启 cmaster0 机, 故必须执行上述命令, 手动启动 ambari-server 服务, 在 cmaster0 命令行下输入下述命令, 即可看到 AmbariServer 进程:

```
[root@cmaster0 ~]# /usr/jdk64/jdk1.7.0_67/bin/jps
32304 Jps
30209 AmbariServer
```



```
[root@cmaster0 ~]# ps -ef | grep AmbariServer
```

这两个命令都用来查看进程，输出较多，编者并未给出其输出。上文已经讲述 AmbariServer 会为用户提供 Web 接口，此 Web 接口就是 AmbariServer 服务提供的，以 iclient0 为例，登录 iclient0 机，打开 FireFox 浏览器并在地址栏输入：

```
http://cmaster0.cloudlab.njupt.edu.cn:8080
```

回车后，可看到图 3-7 所示的 Ambari 登录界面，即表示 Ambari 启动成功。

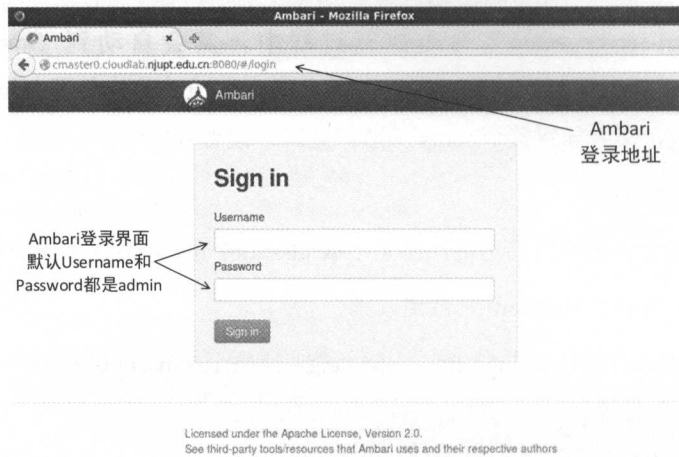


图 3-7 Ambari 登录界面

由于 prelittleCstor 内九台机器都处于同一局域网，这九台机器都可打开此网页，如果某机打不开此页面，则说明前面的 prelittleCstor 集群搭建过程中该机某步骤出错。

其实，只要是同一个网段的机器都可打开此网页，比如图 3-8 为编者笔记本电脑上 FireFox 登录示意图。从这两幅图可以看出，win7 截图清晰度比 CentOS 截图清晰度高出许多。

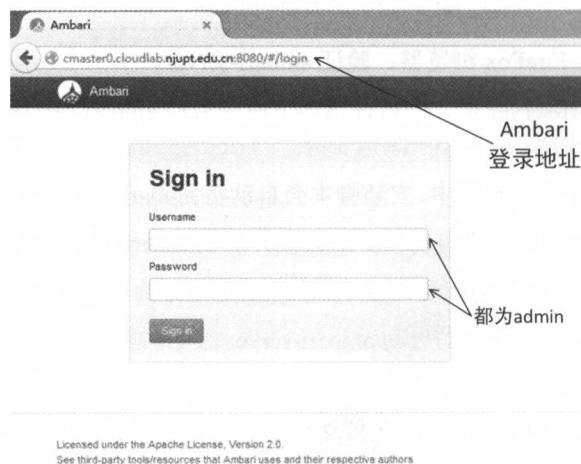


图 3-8 Ambari 登录界面

由于编者所在机构南京邮电大学内部网络为一完整局域网，故此 AmbariServer 服务面向南邮校内所有用户。不过由于 10.X.X.X 为专用网络，加之编者并未开启 VPN，非南邮用户打不开此网址。至此，整个 AmbariServer 部署结束。

综上所述，本部分主要完成将 ambari-server 安装到 cmaster0 上，并对 ambari-server 进行了初始设置。在安装过程中，Ambari 安装包下载自 repo 机。

3.3.6 搭建 littleCstor

古语有言“万事俱备只欠东风”，前面的一切准备都为了这一步部署。当前，已经有了物理集群 prelittleCstor，部署工具 Ambari，所有大数据组件安装包（都存储在 repo 机上）。本步骤的目的是，在 prelittleCstor 上安装所有大数据组件。其过程可以描述为，使用 Ambari 到 repo 机下载所有大数据组件安装包，并将这些安装包安装到 prelittleCstor 中各台机器上。

简单地讲，本部分输入为 prelittleCstor 中所有机器和 repo 上所有大数据组件安装包，输出部署好所有大数据组件的 littleCstor，其中间部署过程全部由 Ambari 负责。

（1）规划“使用 Ambari 部署 HDP”

当前版本的 HDP 包含 15 个大数据组件，编者的规划很简单，安装完整的 HDP，一个组件也不少。不过为尽量确保组件角色清晰，对部分（如 NameNode）主服务，并不部署其对应 HA（High Available）节点。下面为编者规划要领：

部署完整 HDP 但不涉及 HA，cmasterX 为主节点，cslaveX 为从节点，iclient0 为客户节点。

（2）使用 Ambari 部署 HDP 主要步骤

- ①针对所有机器：打通 AmbariServer 到其他机器 SSH 无密钥登录。
 - ②AmbariServer 机：拷贝本机“/root/.ssh”目录下 id_rsa 至桌面。
 - ③AmbariServer 机：启动 ambari-server。
 - ④集群任意机 FireFox 浏览器：打开“AmbariServer:8080”。
 - ⑤Ambari 主页：输入用户名、密码，登录。
 - ⑥Ambari 主页：根据向导建立大数据集群。
- Ambari 主页部署向导注意点：务必修改默认仓库位置、上传 id_rsa。

1. Step1

- ①针对所有机器：打通 AmbariServer 到其他机器（repo 除外）SSH 无密钥登录。

编者的 AmbariServer 机即为 cmaster0, 故上述要求是实现 cmaster0 到下述九台机器无密钥登录:

```
cmaster0、cmaster1、cmaster2、cslave0、cslave1、cslave2、cslave3、cslave4、iclient0
```

读者可能会有疑惑, 为何要实现 cmaster0 到 cmaster0 自身 SSH 无密钥登录, 何为自身登录自身? 有何意义? 又如何实现?

这是编者本人问题, 请读者见谅, 编者暂未购置新机器来独立部署 AmbariServer, 故将 ambari-server 也部署到了 cmaster0。此时, cmaster0 上还要部署大量 HDP 主服务, 而部署 HDP 的节点就必然要部署 ambari-agent, 故 cmaster0 上既要部署 ambari-server 又要部署 ambari-agent。前文已经说明“要实现 Ambari 主节点到 Ambari 从节点 SSH 无密钥登录”, 既然 cmaster0 充当了 ambari-agent 节点, 那肯定要实现 cmaster0 到 Ambari-Server 机 (即 cmaster0 机) 无密钥登录, 也就是实现 cmaster0 到 cmaster0 无密钥登录。至于何为自身登录自身, 从代码角度就是“ssh localhost”与“ssh `hostname`”, 其他读者暂不必关心。

由于编者自身原因给读者理解带来不便, 还请读者见谅。下面具体讲述如何实现 cmaster0 到包含 cmaster0 在内的九台机器 SSH 无密钥登录。

One cmaster0 机 SSH 前期准备

SSH 的密钥文件夹为本地隐藏文件夹.ssh, 下述命令为先备份或删除.ssh 文件, 请读者务必注意, 若集群并不属于个人, 请尽量不要随意删除任何文件。

```
[root@cmaster0 ~]# mv .ssh .sshBak          #如果已经有.ssh 文件,很重要,备份
[root@cmaster0 ~]# rm -rf .ssh               #不重要,删除
```

由于编者的 prelittleCstor 集群刚刚搭建, 暂无任何人使用, 且属于编者个人所有, 故直接删除已有.ssh 文件。

```
[root@cmaster0 ~]# rm -rf .ssh          #编者操作, prelittleCstor 刚建立, 删除
[root@cmaster0 ~]# ssh localhost        #验证 SSH 登录本机是否需要密码
The authenticity of host 'localhost (::1)' can't be established.
RSA key fingerprint is 81:5b:bb:36:2f:6d:41:52:33:8b:f5:48:36:6a:31:8e.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (RSA) to the list of known hosts.
root@localhost's password:
Permission denied, please try again.
root@localhost's password:
Permission denied, please try again.
root@localhost's password:
Permission denied (publickey,gssapi-keyex,gssapi-with-mic,password).
[root@cmaster0 ~]#
```

上述第二条命令为验证当前是否已经实现 SSH 无密钥登录, 很明显, 已经删除了.ssh, 此处必然需要密钥才能登录, 否则不合逻辑。这么做是为了让读者看到, 现在的确要输入密钥, 这样可对比 **Three** 中无密钥情况, 加深理解。

Two cmaster0 机生成公私钥

```
[root@cmaster0 ~]# rm -rf .ssh
[root@cmaster0 ~]# ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Created directory '/root/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa.
Your public key has been saved in /root/.ssh/id_rsa.pub.
The key fingerprint is:
b7:0e:8f:97:34:3c:b4:b4:05:d3:27:f6:a3:16:9f:fb root@cmaster0
The key's randomart image is:
+--[ RSA 2048 ]-----+
|          .            |
|         o + .         |
|        + +           |
|       o o o          |
|      S+. + + o       |
|     .B.o o          |
|    ...=             |
|   =o                |
|  ..o               .E|
+-----+
[root@cmaster0 ~]# ls -all .ssh
total 16
drwx----- 2 root root 4096 Apr 22 15:14 .
dr-xr-x--- 4 root root 4096 Apr 22 15:14 ..
-rw----- 1 root root 1675 Apr 22 15:14 id_rsa
-rw-r--r-- 1 root root 395 Apr 22 15:14 id_rsa.pub
```

图 3-9 cmaster0 机密钥生成

如图 3-9 所示，在 cmaster0 上执行“ssh-keygen”命令生成公私钥。第一个提示是询问将公私钥文件存放在哪，直接回车，选择默认位置。

第二个提示是请求用户输入密钥，既然操作的目的是实现 SSH 无密钥登录，故此处必须使用空密钥，所谓的空密钥指的是直接回车，不是空格，更不是其他字符。此处请读者务必直接回车，使用空密钥。第三个提示是要求用户确认刚才输入的密钥，既然刚才空密钥（直接回车即空），那现在也应为空，直接回车即可。

最后，我们通过命令“ls -all /root/.ssh”查看到，SSH 密钥文件夹.ssh 目录下的确生成了两个文件 id_rsa 和 id_rsa_pub，这两个文件都有用，其中公钥用于加密，私钥用于解密。中间的 rsa 表示算法为 RSA 算法，读者也可使用 DSA 算法，SSH 本身理论知识较多，请读者参考相关资料^[3]，这里不再讲述。

Three 将 cmaster0 机生成的公钥追加到上述九台机器的 “/root/.ssh/authorized_keys” 文件中

按要求，需要将公钥文件 “/root/.ssh/id_rsa.pub” 追加到上述九台机器的 SSH 认证 “/root/.ssh/authorized_keys” 中。这九台机中有 cmaster0 本机，不过，实际上，操作步骤几乎相同，下面仅以 cmaster0 和 cslave0 为例，其他机器请参照 cslave0。

- 将 cmaster0 公钥追加到本机认证文件

在 cmaster0 上，首先通过 “ssh localhost” 和 “ssh `hostname`” 命令查看此时是否可以无密钥登录本机，其实此步可不必执行，这么做是为了说明当前 SSH 登录的确需要输

入密钥, 以对比无密钥情况。命令输出中, 当提示是否连接时, 请回答“yes”, 当提示输入密码时, 直接回车, 放弃登录。图中标记星号的“cat /root/.ssh/id_rsa.pub >> /root/.ssh/authorized_keys”命令为操作核心语句, 此命令完成将 cmaster0 机公钥文件 id_rsa.pub 追加到 cmaster0 机(即自身)认证文件 authorized_keys。操作完此步后, 再次执行“ssh localhost”和“ssh ‘hostname’”时, 就不需要输入密钥了(图 3-10)。

```
[root@cmaster0 ~]# ssh localhost
The authenticity of host 'localhost (::1)' can't be established.
RSA key fingerprint is a2:3f:04:d6:8e:1e:37:8b:bd:7d:b4:e2:4c:69:75:29.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (RSA) to the list of known hosts.
root@localhost's password:
Permission denied, please try again.
root@localhost's password:
Permission denied, please try again.
root@localhost's password:
Permission denied (publickey,gssapi-keyex,gssapi-with-mic,password).
[root@cmaster0 ~]# ssh 'hostname'
The authenticity of host 'cmaster0 (192.168.170.133)' can't be established.
RSA key fingerprint is a2:3f:04:d6:8e:1e:37:8b:bd:7d:b4:e2:4c:69:75:29.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'cmaster0,192.168.170.133' (RSA) to the list of known hosts.
root@cmaster0's password:
Permission denied, please try again.
root@cmaster0's password:
Permission denied, please try again.
root@cmaster0's password:
Permission denied (publickey,gssapi-keyex,gssapi-with-mic,password).
[root@cmaster0 ~]# cat /root/.ssh/id_rsa.pub >> /root/.ssh/authorized_keys
[root@cmaster0 ~]#
[root@cmaster0 ~]# ssh localhost
Last login: Fri Jan 15 08:39:15 2016
[root@cmaster0 ~]# exit
logout
Connection to localhost closed.
[root@cmaster0 ~]# ssh 'hostname'
Last login: Sat Apr 23 01:39:17 2016 from localhost
[root@cmaster0 ~]# exit
logout
Connection to cmaster0 closed.
[root@cmaster0 ~]#
```

验证是否可以无密钥登录localhost(本机自己)

必须输入yes

输入root用户的密码才可登录,不过此处我们不输入任何字符,直接回车,放弃登录

认证失败, 放弃登录

验证是否可以无密钥登录cmaster0(网络内自己)

必须输入yes

输入root用户的密码才可登录,不过此处我们不输入任何字符,直接回车,放弃登录

认证失败, 放弃登录

★ 无密钥认证核心操作: 将公钥追加到认证文件

再次验证是否可以无密钥登录localhost(本机自己)

已无需密钥

退出自己

再次验证是否可以无密钥登录cmaster0(网络内自己)

已无需密钥

退出自己

图 3-10 实现 cmaster0 无密钥登录 cmaster0

关于“本机自己”和“网络中自己”的区别, 请读者参阅环回地址和普通地址的区别。本书不是讲解计算机网络的专业书籍, 故此处不做深入讲解。

从上述过程中可以看出, 将 cmaster0 公钥追加到 cmaster0 认证文件后, 即可实现 cmaster0 到 cmaster0 无密钥登录, 同样, 将 cmaster0 公钥追加到 cslave0 认证文件, 即可实现 cmaster0 到 cslave0 无密钥认证, 以下内容讲述实现 cmaster0 无密钥登录 cslave0。

- 将 cmaster0 公钥追加到 cslave0 机认证文件中

如下操作中, 先通过执行“ssh cslave0.cloudlab.njupt.edu.cn”, 查看当前 cmaster0 是否已经可以无密钥登录 cslave0, 结果正如图 3-11 所示, 不可以。

由于 cslave0 上并不存在 cmaster0 的 SSH 公钥文件, 先将 cmaster0 机的 SSH 公钥文件 id_rsa.pub 拷贝至 cslave0 机。编者使用了一个看似高端的 scp 命令, 其实此命令效果和使用 U 盘将 cmaster0 上文件拷贝至 cslave0 上没什么区别, 不知道 scp 的读者, 请查阅相关资料^[4]。不过当读者使用 U 盘拷贝时, 由于.ssh 为隐藏文件, 读者须先将文件拷出隐藏

文件夹，然后才可查看到文件。注意下述命令都在 cmaster0 上执行。

```
[root@cmaster0 ~]# ssh cslave0
root@cslave0's password:
Permission denied, please try again.
root@cslave0's password:
Permission denied, please try again.
root@cslave0's password:
Permission denied (publickey,gssapi-keyex,gssapi-with-mic,password).
[root@cmaster0 ~]#
[root@cmaster0 ~]# scp /root/.ssh/id_rsa.pub root@cslave0:~/
root@cslave0's password:
id_rsa.pub
[root@cmaster0 ~]#
```

从cmaster0登录cslave0,读者务必将cslave0
换成cslave0.cloudlab.njupt.edu.cn

目前的确定需要密钥
直接回车,放弃登录

认证失败,未登录

★ 核心1: 使用scp命令将
cmaster0公钥文件远程拷贝至cslave0

100% 395 0.4KB/s 00:00

输入cslave0机root密码,完成远程拷贝

图 3-11 将 cmaster0 公钥远程拷贝至 cslave0

完成拷贝后，请登录 cslave0 机，下述命令中，先通过“ll”确认文件是否已经拷贝过来（图 3-12）。接着查看 cslave0 上是否存在.ssh 目录，如果不存在，手工新建此目录。当存在此文件夹时，读者可试着执行“cat...”操作，若命令执行不成功，删除.ssh 目录，接着新建.ssh，再次执行“cat...”。

命令“cat /root/.ssh/id_rsa.pub >> /root/.ssh/authorized_keys”为本次操作核心命令，该命令完成将 cmaster0 拷贝过来的 id_rsa.pub 文件追到 cslave0 的 ssh 认证文件 authorized_keys 中（图 3-12）。该认证添加后，cmaster0 远程登录 cslave0 时，就不需要输入密钥了。其实该操作和实现 cmaster0 登录 cmaster0 的核心语句一样，都是将公钥添加到认证文件里，不同的是，公钥文件是哪台机器的公钥文件，认证文件又是哪台机器的认证文件。集群环境复杂，请读者务必弄清楚当前身处何机，又是何用户（你是谁，你在哪）。

回到 cmaster0，在 cmaster0 命令行再次执行“ssh cslave0.cloudlab.njupt.edu.cn”，从图 3-13 可以看出，此次登录时，已不再需要输入密钥。

```
[root@cslave0 ~]# ll
total 68
-rw-r--r-- 1 root root 3326 Nov 13 09:05 anaconda-ks.cfg
-rw-r--r-- 1 root root 395 Apr 23 03:01 id_rsa.pub
-rw-r--r-- 1 root root 41800 Nov 13 09:05 install.log
-rw-r--r-- 1 root root 9154 Nov 13 09:02 install.log.syslog
[root@cslave0 ~]# ls -all ~/.ssh
ls: cannot access /root/.ssh: No such file or directory
[root@cslave0 ~]#
[root@cslave0 ~]# ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Created directory '/root/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
[root@cslave0 ~]#
[root@cslave0 ~]# cat id_rsa.pub >> .ssh/authorized_keys
[root@cslave0 ~]#
```

查看刚才是否已将文件拷贝至本机

就是这个文件,刚scp远程
拷贝过来的cmaster0机公钥

确定本机是否有该文件夹
有,最好;无,使用ssh-keygen新建

尽量使用ssh-keygen方式,不使用mkdir

直接回车

大量省略

★ 核心2
将cmaster0的公钥文件追加
到本机(cslave0)的认证文件里

图 3-12 将从 cmaster0 拷来的公钥追加到 cslave0 认证文件

```
[root@cmaster0 ~]# ssh cslave0
Last login: Sat Apr 23 03:03:26 2016 from cmaster0
[root@cslave0 ~]#
[root@cslave0 ~]# exit
logout
Connection to cslave0 closed.
[root@cmaster0 ~]#
```

再次验证是否可无密钥从cmaster0登录到cslave0
读者务必将cslave0换成cslave0.cloudlab.njupt.edu.cn
的确不需要输入密钥
退出cslave0
又回到了cmaster0上

图 3-13 确认 cmaster0 到 cslave0 无密钥登录

至此，已成功实现了：

- 从 cmaster0 机 SSH 无密钥登录 cmaster0 机。
- 从 cmaster0 机 SSH 无密钥登录 cslave0 机。

按要求，须实现 cmaster0 到 prelittleCstor 内所有机器（repo 除外）SSH 无密钥登录。

除去 cmaster0 本身和 cslave0 外，还剩下如下 7 台：

cmaster1、cmaster2、cslave1、cslave2、cslave3、cslave4、iclient0

实现 cmaster0 到这 7 台机器 SSH 无密钥登录操作和实现 cmaster0 到 cslave0 操作一模一样，请读者参照上述步骤，自行完成。注意，操作时正常这 7 台机都会存在“/root/.ssh/”目录，读者不必删除，直接执行“cat /root/.ssh/id_rsa.pub >> /root/.ssh/authorized_keys”即可。

机器 cmaster0 到九台机 SSH 无密钥都打通后，为确保无误，请手工确认已无需密钥。将如下内容写入文件 machines 中。

```
cmaster0.cloudlab.njupt.edu.cn
cmaster1.cloudlab.njupt.edu.cn
cmaster2.cloudlab.njupt.edu.cn
cslave0.cloudlab.njupt.edu.cn
cslave1.cloudlab.njupt.edu.cn
cslave2.cloudlab.njupt.edu.cn
cslave3.cloudlab.njupt.edu.cn
cslave4.cloudlab.njupt.edu.cn
iclient0.cloudlab.njupt.edu.cn
```

然后在 cmaster0 上执行如下脚本：

```
for x in `cat machines`; do echo $x ; ssh $x true ; done ;
```

此脚本会遍历 machines 文件中每一台机器，接着输出并验证是否可 SSH 至此机器。此脚本看似高端，实际上就是“ssh cslave0.cloudlab.njupt.edu.cn”执行了九次而已（当然机器名不断改变），执行过程如图 3-14 所示。

至此，已成功实现 cmaster0 机 SSH 无密钥登录其他九台机器。步骤①结束，此步骤大量涉及 SSH 操作，可 SSH 本身就包含许多概念，不了解 SSH 理论，很难看懂本实例，限于篇幅，本书并未涉及 SSH 理论，若读者需要深入 SSH，请参阅文献[3]。

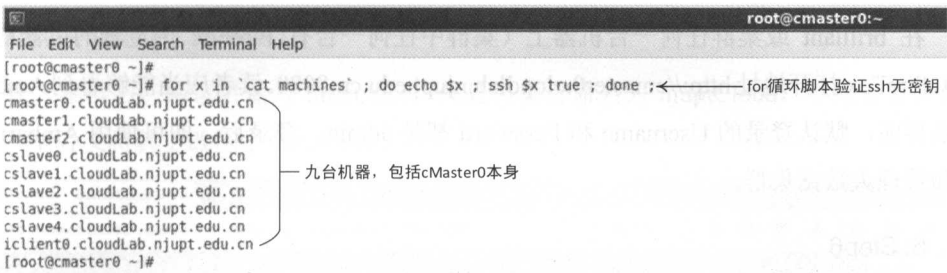


图 3-14 验证 SSH 无密钥登录

2. Step2

②AmbariServer 机：拷贝本机 “/root/.ssh” 目录下 id_rsa 至桌面。

当前的 AmbariServer 机即为 cmaster0，故登录 cmaster0 后，将“/root/.ssh/id_rsa”拷贝至桌面即可（图 3-15）。

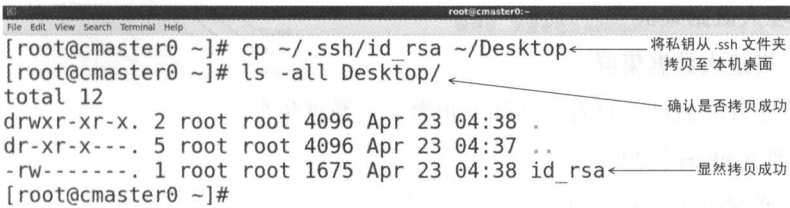


图 3-15 将 cmaster0 机 SSH 私钥拷贝至桌面

图 3-15 中命令 “cp ~/.ssh/id_rsa ~/Desktop” 即完成将 id_rsa 文件拷贝至桌面，注意，由于操作在 cmaster0 上执行，故拷贝后，id_rsa 在 cmaster0 机的桌面上。

3. Step3

③AmbariServer 机：启动 ambari-server。

实际上，在 cmaster0 开机时 ambari-server 服务已经自动启动，不过不管有没有启动，我们都可以重启 ambari-server 服务。编者的 AmbariServer 机即为 cmaster0，下述命令在 cmaster0 上重启 ambari-server：

```
[root@cmaster0 ~]# service ambari-server restart
```

4. Step4

④集群任意机 FireFox 浏览器：打开 “AmbariServer:8080”。

5. Step5

⑤Ambari 主页：输入用户名、密码，登录。

在 brilliant 或集群任何一台机器上 (集群中任何一台有 FireFox 浏览器的机器均可以), FireFox 打开地址 <http://cmaster0.cloudlab.njupt.edu.cn:8080>。读者应当能够看到 Ambari 登录界面, 默认登录的 Username 和 Password 都是 admin。登录后, 即可使用 Ambari 搭建和管理大数据集群。

6. Step6

前面的一切努力都是为了最终的部署, 请读者静下心来, 跟随编者收获果实。步骤⑥内容为:

⑥Ambari 主页: 根据向导建立大数据集群。

在 brilliant (或集群中任何一台有 FireFox 的机器) 上, FireFox 打开地址 <http://cmaster0.cloudlab.njupt.edu.cn:8080>。此页面为 Ambari 主服务登录页面, 页面主要包含以下功能。

- 管理大数据集群
 - 搭建大数据集群
 - 监管 (启动、查看、关闭、报警) 大数据集群
- 管理 Ambari 用户
 - 增删改 Ambari 用户

通过 Ambari 页面, 不仅可以搭建大数据集群, 还可以监控大数据集群, 比如当集群中某 DataNode 突然停止时, Ambari 会发出警报并向 Ambari 管理员发送报警邮件。

正常情况下, 应当由公司运维部负责监控大数据集群。而在部门内部, 不可能只要求某一位运维工程师完成管理 Ambari, 这就必然涉及多个 Ambari 账户和账户权限问题。Ambari 页面上提供了这样的用户管理和授权功能, 通过设置, 甚至可以开启来宾账户 (权限很受限), 提供第三方客户页面查看功能。

由于 prelittleCstor 刚新建完成, 还并未在其上部署 HDP, 下面讲述如何使用 Ambari 搭建大数据集 littleCstor。

在 FireFox 中打开并登录 Ambari。

请用户使用 FireFox 打开地址 <http://cmaster0.cloudlab.njupt.edu.cn:8080>, 用户可以使用集群中任何一台机器, 不过该机需要开启桌面且有 FireFox。由于编者集群中, 所有机器均未开启桌面, 故使用集群外的 brilliant 机, 该机实质上是编者笔记本电脑, 其 OS 为 Win7, 且装有最新版的 FireFox。读者可以采用编者方式, 使用自己电脑来执行一切远程操作, 也可以使用集群中机器, 比如继续使用 cmaster0 进行操作。

FireFox 打开 Ambari 地址后, 初始 Username 和 Password 均为 admin, 接着点击链接 “Launch Install Wizard”, 进入下一环节。

1) Get Started (图 3-16)

输入集群名，请读者输入 littleCstor，此处将集群命名为 njuptCloud，不过集群搭建好后，编者又将其更改回了 littleCstor。

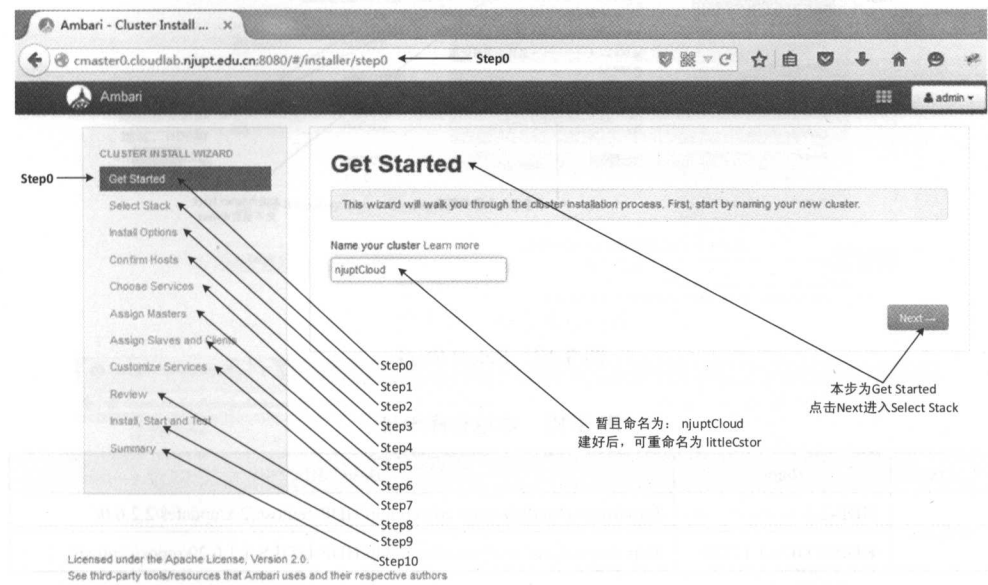


图 3-16 Get Started

2) Select Stack

本部分需要配置两个参数，第一个是选定 HDP 版本，第二个是指定 HDP 仓库位置。本部分的配置非常重要，特别是指定 HDP 仓库位置，如果不手工指定 HDP 仓库位置，当需要使用大数据安装包时，Ambari 会到默认地址 Hortonworks 公司 HDP 仓库去下载大数据安装包。可是该默认地址（Hortonworks 公司 HDP 仓库）网速受限，中国用户很难能在 15 分钟内下载近 4G 安装包，最终将导致超时失败。

①选择 HDP（Hortonworks Data Platform）版本，此处选择 2.2，请读者尽量选择当前最新版（图 3-17）。

②打开“Advanced Repository Options”标签，取消已选中的“redhat5”、“suse11”、“ubuntu12”，只保留“redhat6”。接着，将 HDP-2.2 和 HDP-UTILS-1.1.0.20 值分别修改为表 3-13 中对应值。

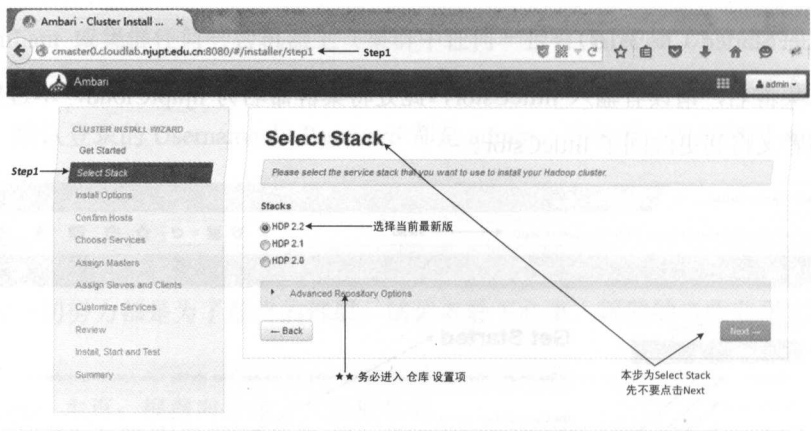


图 3-17 Select Stack

表 3-13 本地仓库地址

OS	Name	Base URL
redhat6	HDP-2.2	http://repo.cloudlab.njupt.edu.cn/hdp/HDP/centos6/2.x/updates/2.2.6.0/
	HDP-UTILS-1.1.0.20	http://repo.cloudlab.njupt.edu.cn/hdp/HDP-UTILS-1.1.0.20/repos/centos6/

有的读者可能会奇怪，集群中各机 OS 均为 CentOS6，为何选择 redhat6，那是因为 CentOS 和 redhat 是一个体系，其实质上就是 redhat 依照开源规范释放的源码编译而成的。再次强调，请读者务必配置 Ambari 使用本地仓库。

图 3-18 和图 3-19 为配置 Ambari 使用本地仓库的过程，请读者务必配置 Ambari 使用本地仓库。完成配置后，点击“Next”，进入图 3-20 所示界面。



图 3-18 只选中 redhat6

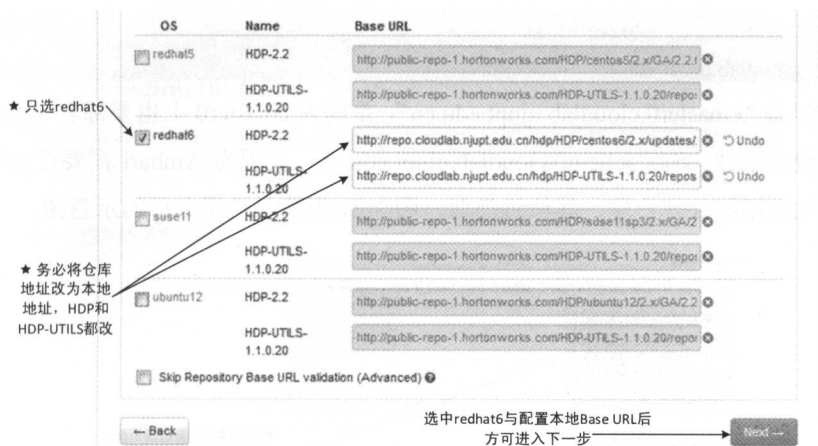


图 3-19 配置本地仓库

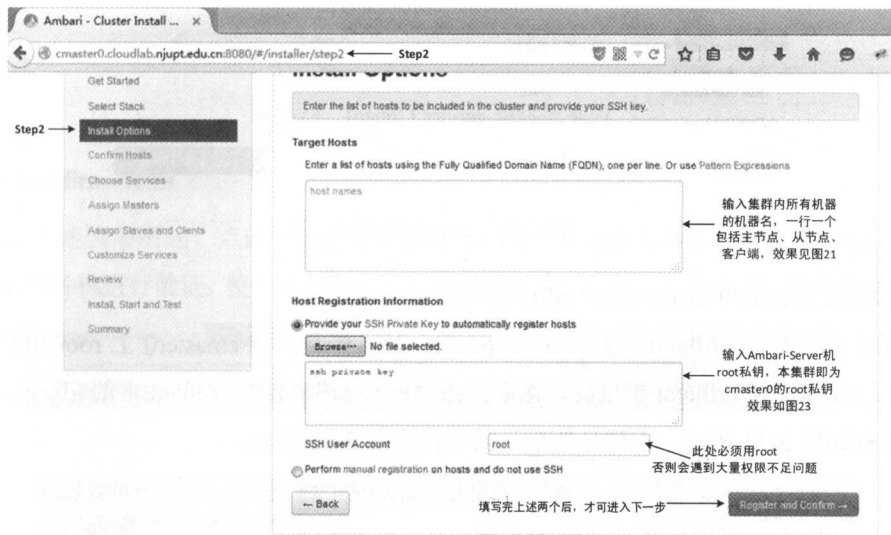


图 3-20 Install Options

3) Install Options

本部分也分为两步，第一步，输入集群中 `prelittlecstor` 所有机器的机器名；第二步，输入 `cmaster0` 机 `root` 用户的 SSH 私钥。

①将如下机器名全部输入“Target Hosts”选项框：

```
cmaster0.cloudlab.njupt.edu.cn
cmaster1.cloudlab.njupt.edu.cn
cmaster2.cloudlab.njupt.edu.cn
cslave0.cloudlab.njupt.edu.cn
cslave1.cloudlab.njupt.edu.cn
cslave2.cloudlab.njupt.edu.cn
cslave3.cloudlab.njupt.edu.cn
```

cslave4.cloudlab.njupt.edu.cn

iclient0.cloudlab.njupt.edu.cn

之所以有“cmaster0.cloudlab.njupt.edu.cn”，是因为 cmaster0 上也要部署 ambari-agent 和部分大数据组件；有“iclient0.cloudlab.njupt.edu.cn”是因为 Ambari 需要在 iclient0 上部署所有大数据组件客户端，其他机类似，图 3-21 为输入机器名后的示意图。

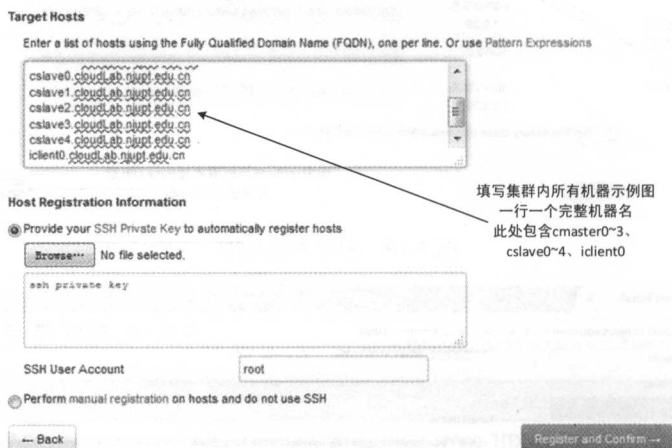


图 3-21 Install Options

②输入 cmaster0 机 root 用户 SSH 私钥。

由于编者是在 brilliant 上打开 Firefox 的，故操作是：将 cmaster0 上 root 用户 SSH 私钥 id_rsa 拷贝至 brilliant 机桌面，接着点击“Browse”，选中 brilliant 上的 id_rsa，具体操作过程如图 3-22 所示。上传私钥后，效果图如图 3-23 所示。



图 3-22 选择 cmaster0 上 root 用户 SSH 私钥 id_rsa

图 3-23 为 Install Options 最终效果图，确定无误后，点击图中的“Register and Confirm”，进入“Confirm Hosts”。

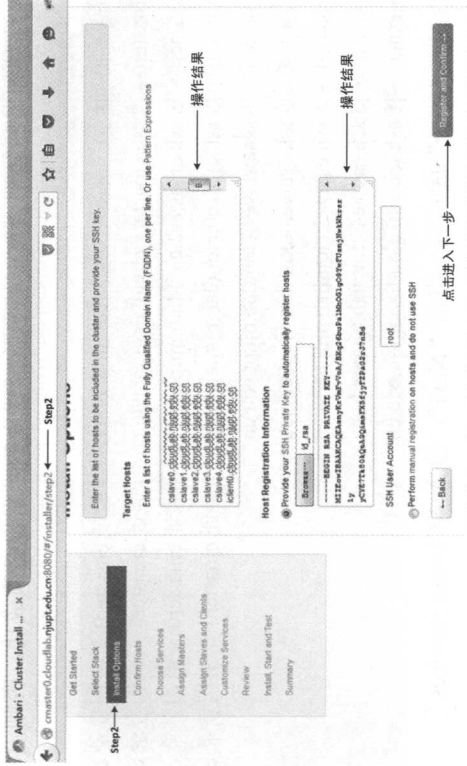


图 3-23 Install Options 最终效果图

4) Confirm Hosts

正如上述内容所述，点击“Register and Confirm”方可进入本步。本步 Ambari 会有 id_rsa 到各机进行验证，整个验证过程会持续较长时间，其过程如图 3-24、图 3-25 所示。

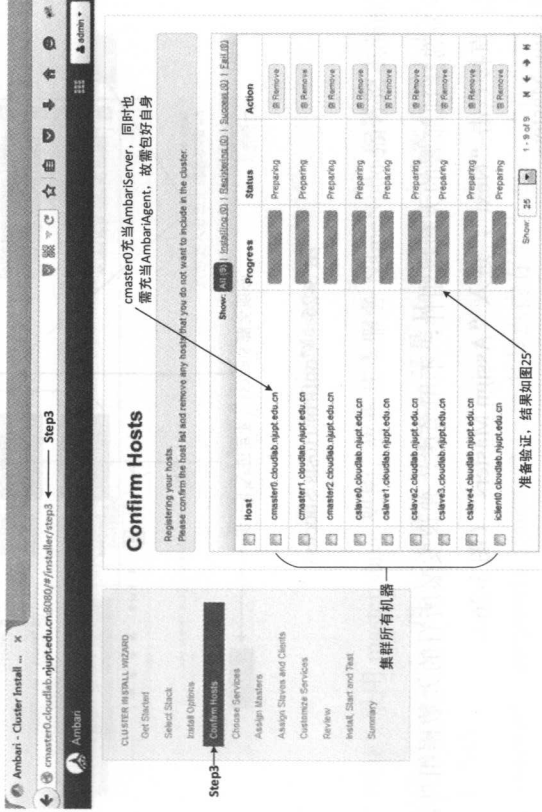


图 3-24 Confirm Hosts Preparing

如果“Confirm Hosts”不成功，请读者再次确认 SSH 是否已成功打通。请确定“id_rsa”是否是 AmbariServer 机（cmaster0）的“id_rsa”。正常情况下，只要能够确保如下命令执行时，均不输入密码，本步就应当能够成功执行。如果还是不成功，则一定是在前面搭

建 `prelitttleCstor` 过程中，某步没有执行或没有成功执行，比如未添加域名映射，未修改机器名等。HDP 部署条件非常苛刻，比如 `cslave0` 的 OS 为 CentOS6.7，而 `cslave1` 的 OS 为 CentOS6.6，这种细微的差别，都可能导致部署失败。本步必须成功，否则没有机会进入下一步。

```
[root@cmaster0 ~]$ ssh cmaster0.cloudlab.njupt.edu.cn #验证 cmaster0 无密钥登录 cmaster0,root
[root@cmaster0 ~]$ ssh cmaster1.cloudlab.njupt.edu.cn #验证 cmaster0 无密钥登录 cmaster1,root
[root@cmaster0 ~]$ ssh cmaster2.cloudlab.njupt.edu.cn #验证 cmaster0 无密钥登录 cmaster2,root
[root@cmaster0 ~]$ ssh cslave0.cloudlab.njupt.edu.cn #验证 cmaster0 无密钥登录 cslave0,root
[root@cmaster0 ~]$ ssh cslave1.cloudlab.njupt.edu.cn
[root@cmaster0 ~]$ ssh cslave2.cloudlab.njupt.edu.cn
[root@cmaster0 ~]$ ssh cslave3.cloudlab.njupt.edu.cn
[root@cmaster0 ~]$ ssh cslave4.cloudlab.njupt.edu.cn
[root@cmaster0 ~]$ ssh iclient0.cloudlab.njupt.edu.cn
```

待“Confirm Hosts”成功后，点击“Next”，进入“Choose Services”。



图 3-25 Confirm Hosts Success

5) Choose Services

本步为选择预部署的大数据组件，不用考虑，建议安装所有的大数据组件，即什么都不用做，直接点击“Next”，进入“Assign Masters”（图 3-26）。

6) Assign Masters

大数据组件中，有些组件采用 master/slave 架构，有些则采用 client/server 模式，本步骤很简单，将采用 master/slave 架构的 master 服务和采用 client/server 模式的 server 服务，全都分配到主节点上（图 3-27）。

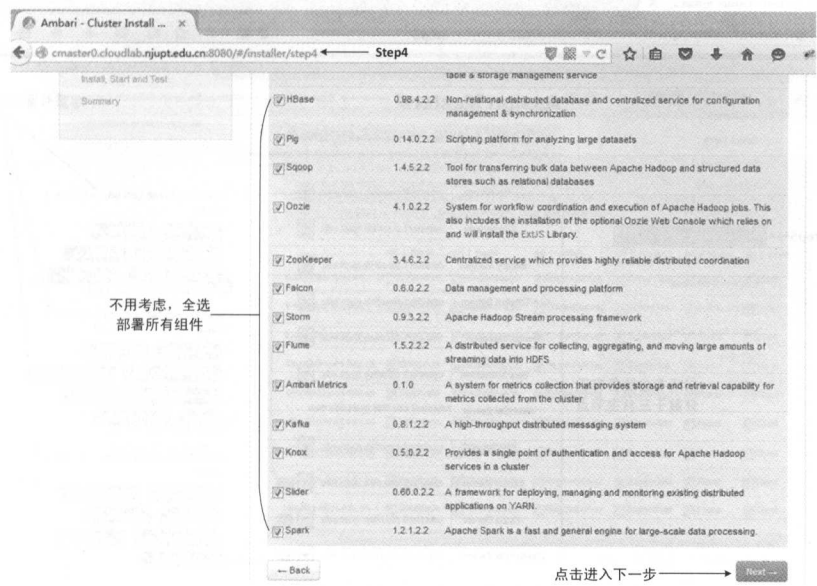


图 3-26 选择要部署的大数据组件

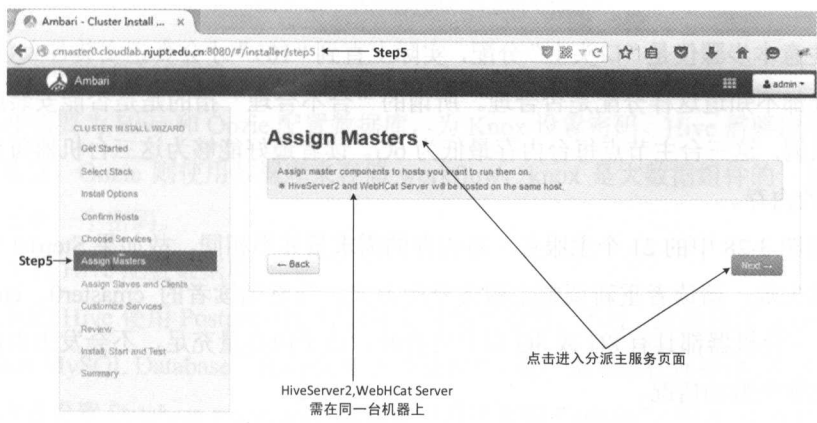


图 3-27 Assign Masters

比如 HDFS 采用 master/slave 架构, 故将其主服务 NameNode 部署到主节点上; 再比如 Hive 采用 client/server 模式(实质上其 server 又是 Hadoop 的一个客户端), 故将其 server 端部署于主节点。

由于 cmaster0、cmaster1、cmaster2 三台主节点内存均为 6G, 故在分配主服务时, 请尽量将它们均匀分散到三台节点上。特别地, ZooKeeper 服务本身至少需要三个节点, 而此处正好三个主节点, 故这三台机器上每台都要部署 ZooKeeper。图 3-28 为编者的主服务分配状态图, 在均匀分散主服务后, 继续点击“Next”, 进入“Assign Slaves and Clients”。

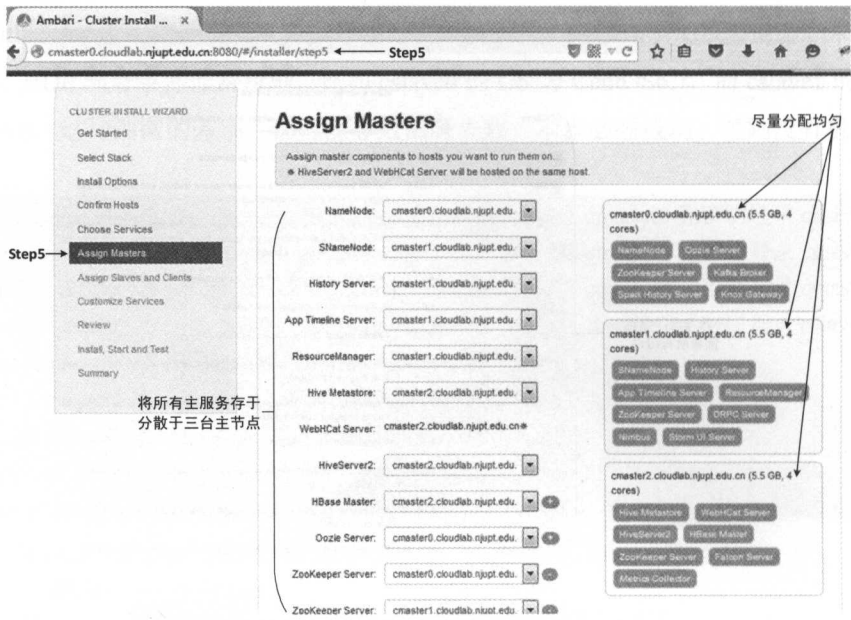


图 3-28 Assign Masters 结果图

请注意本步骤仅是“口头”分配，实际上直到“10”才会真正安装各大数据组件，故此处谁都不知道这样分配是否合理。所谓的“合不合理”指的是是否能安装成功，编者一再强调，这三台主节点每台内存最低为 6G。读者最好能够为这三台机器每台都配置 8G 以上的内存。

显然图 3-28 中的 21 个主服务，对内存的需求量并不相同，故如果 Step10 中提示主服务安装失败，请读者重新调整主服务分配方式。当然当读者的 cmaster0、cmaster1、cmaster2 三台机器都具有 8G 或 8G 以上内存时。由于内存量充足，不会发生由内存不足而引起部署失败的情况。

7) Assign Slaves and Clients

本步为分配各大数据组件从服务和客户端，请读者确保不要将从服务和 Client 模块再分到 cmaster0、cmaster1、cmaster2 上，这三台机器已经很“累”了，承受不住更多服务。对于如下的 5 台 slave 机，除了 Client 服务外，其他服务务必全部选中。本来就是打算让这五台充当 slave 角色，肯定全选。

```
cslave0、cslave1、cslave2、cslave3、cslave4
```

对于 iclient0 这台机器，只安装 Client 服务，其他不安装（图 3-29）。

在本章开始，编者已做出完整规划，cmasterX 上全部（且只）部署主服务，cslaveX 上全部（且只）部署从服务，iclientX 上全部（且只）部署从服务。Step6、Step7 的分配原则完全遵循部署规划，无可挑剔。分配好后，继续点击“Next”，进入“Customize

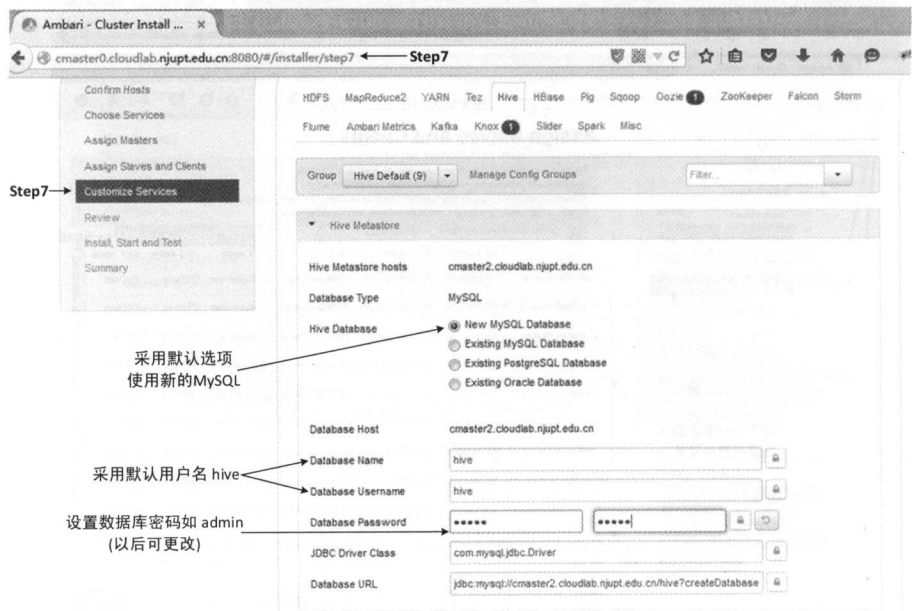


图 3-30 Customize Services Hive

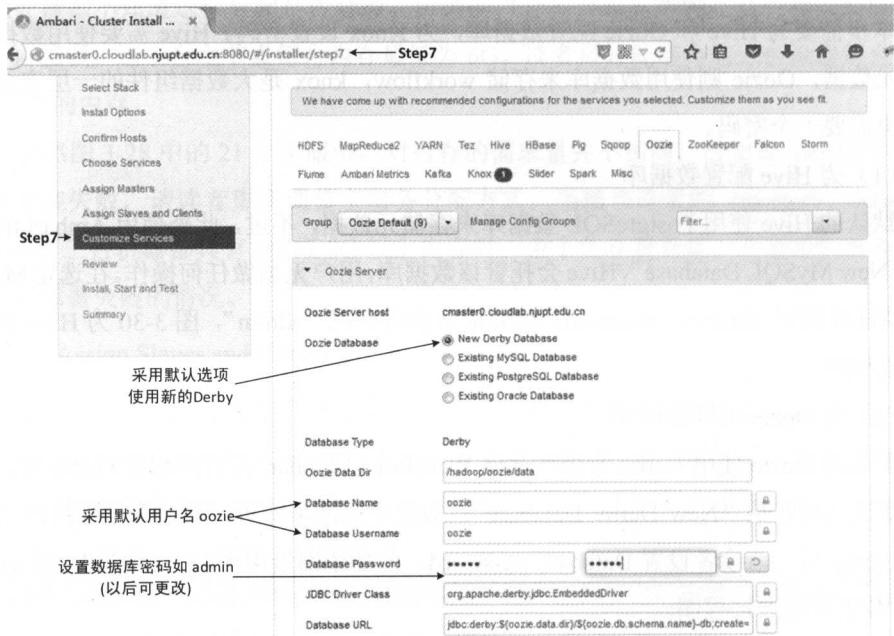


图 3-31 Customize Services Oozie

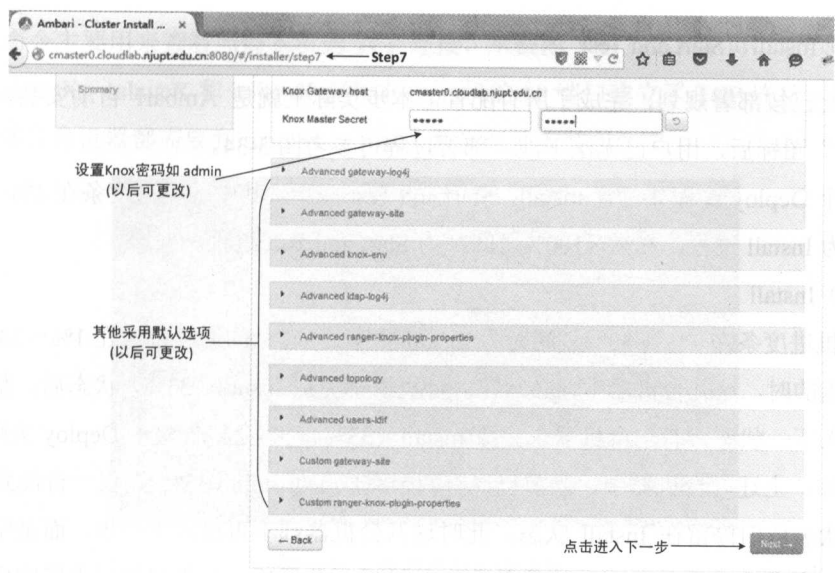


图 3-32 Customize Services Knox

待上述三个必配项配置完成后，继续点击“Next”，进入“Review”。

9) Review

本步主要罗列之前做出的一切配置，并询问用户“是否确认”，请读者将 Review 内所有内容都复制下来，这些内容是所有大数据组件统计性信息（图 3-33）。

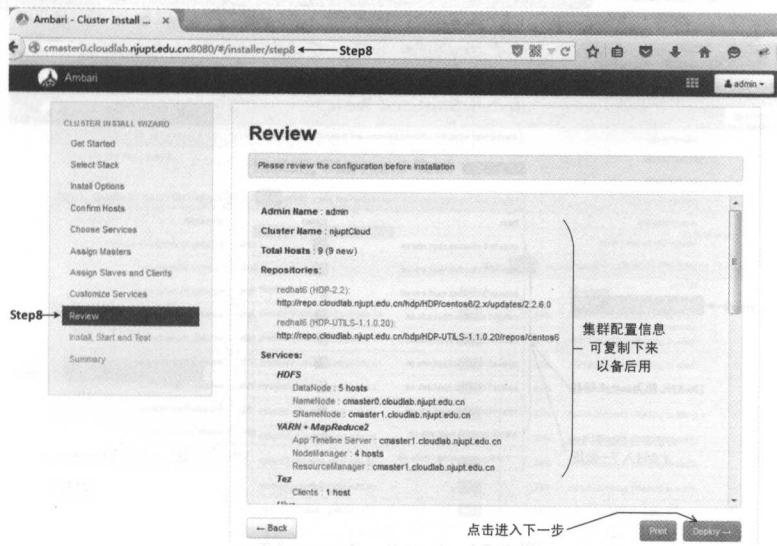


图 3-33 Review

待确定配置无误后，继续点击“Deploy”，进入“Install, Start and Test”。

10) Install, Start and Test

至此已按部署规划，完成了所有配置，本步实际上就是 Ambari 自动安装，当点击“Deploy”图标后，用户已无法干涉，部署过程可参考图 3-34。

整个 Deploy 过程共包含 Install, Start and Test 两步，其中各机进度条在 1%~33%之间时，为 Install 过程，33%~100%之间，为 Start and Test 过程。

(1) Install

各机进度条在 1%到 33%之间时为 Install 状态，此时总的进度条也在 1%~33%之间。需要指出的时，这九台机器中，必须在每台机器都完成 Install (33%) 状态后，整个部署才会进入下一状态，有一台机器未完成 Install (33%)，都会宣告整个 Deploy 失败。

比如，上述九台机器中，当前已经有八台完成了 Install (33%)，仅一台没到 33%，则整个状态依旧停留在 Install 状态。此时这八台机器都不再进入下一步，而是空等最后一台机器完成 Install (33%)。在空等 15 分钟后，若最后一台机器依旧未能完成 Install (33%)，则整个部署过程随之失败。也就是，当集群中有机器未达到 33%时，哪怕是仅有一台，整个部署过程都会失败。

这就是编者为何一再强调“建立本地仓库，将所有大数据组件安装包全都预先下载到本地仓库里”的原因了，假设 cslave1 上要安装全部大数据组件，试问 cslave1 怎么可能在 15 分钟内下载多达 4G 的安装文件。总之，不建立本地仓库，部署不会成功。

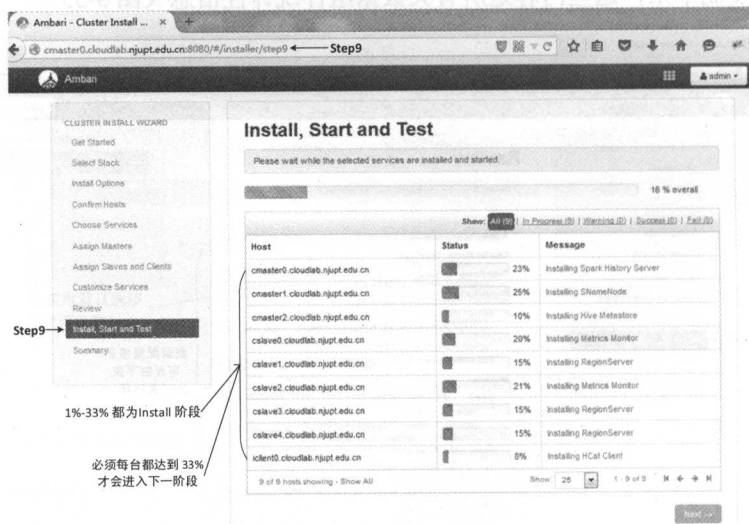


图 3-34 Install, Start and Test

整个安装过程相当漫长，请读者耐心等待。在安装过程中（33%之前），读者可在 repo 机上执行如下命令，其中“10.10.201.115”为 repo 机 IP 地址：

```
[root@repo ~]$ netstat -alpn | grep 10.10.201.115
```

该命令主要用来查看当前本机的 TCP 连接，从截图（图 3-35）中可以看到，当前所有的 `cmasterX`，`cs slaveX` 和 `iclientX` 都和本 `repo` 机的 80 端口建立了 TCP 连接，这是因为集群中这九台机器都需要到 `repo` 机上下载大数据组件安装包。

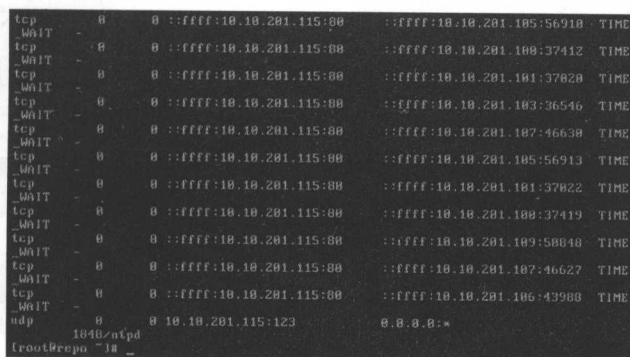


图 3-35 查看当前网络连接

如果不预先将所有大数据组件下载到本地仓库机 repo 里(并将 Ambari 指向该仓库),放任 Ambari 到 Hortonworks 的仓库去下载这些大数据组件安装包,网速太慢,基本上不会下载成功,最终结果是超时失败。试想,即使是迅雷下载这高达 4G 的安装包,都要 2 小时,更何况是 Ambari (内部使用 Python 版的 Wget)。

正如前文所述，Install 过程持续到 33%，并且各机均完成该状态，整个 Deploy 才会进入“Start and Test”（图 3-36）。

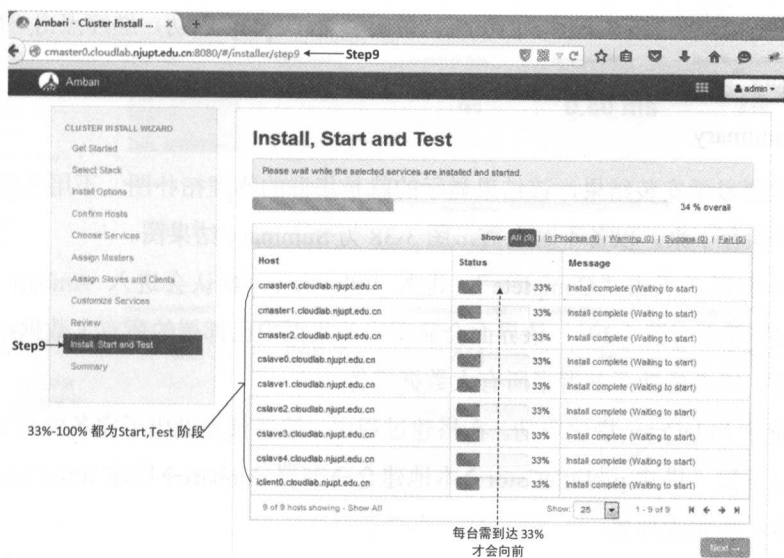


图 3-36 Install, Start and Test

(2) Start and Test

33%~100%均为“Start and Test”状态，其基本步骤是“启动某服务→测试某服务→启动下一个服务→测试下一个服务...”。整个启动和测试过程都由 Ambari 自动完成，用户无须干预，只要 Install 成功，正常本步骤都会成功。实际上，即使不成功，暂时也无法干涉，因为整个“10)”都由 Ambari 自动 Deploy。无须做任何配置，图 3-37 为“Start and Test”执行成功时的状态图。

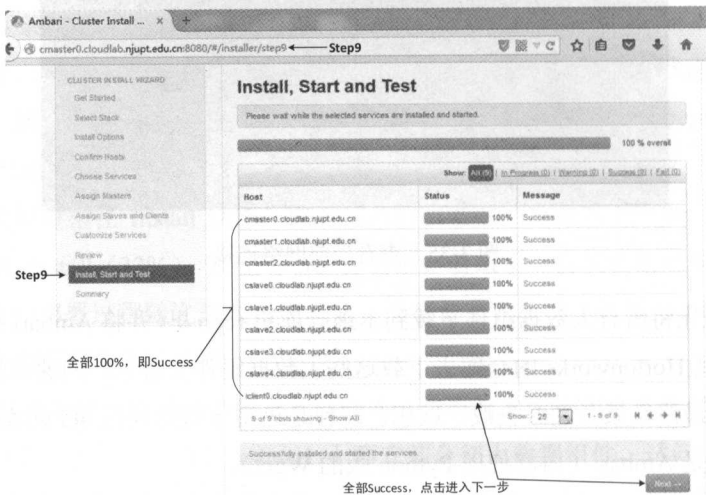


图 3-37 Install,Start and Test Success

图 3-37 中，各进度条已达到 100%，说明整个 Deploy 成功，继续点击“Next”，进入“Summary”。

11) Summary

本步主要显示安装结果，该结果显示的就是集群的物理拓扑图，不用考虑，将所有结果复制并保存下来，以备不时之需，图 3-38 为 Summary 结果图。

确定本步后，点击“Complete”，进入 littleCstor。默认会进入 Ambari 的主面板“Dashboard”界面（图 3-39），该界面会显示当前集群中已部署的所有大数据组件。从图中可以看出，编者成功了部署了所有大数据组件。

至此整个 littleCstor 搭建成功。在搭建过程中，编者预先做出了完备的部署规划，然后在此规划下按“搭建 prelittleCstor→本地建仓→部署 Ambari→搭建 littleCstor”这四个步骤，一步一步完成部署。

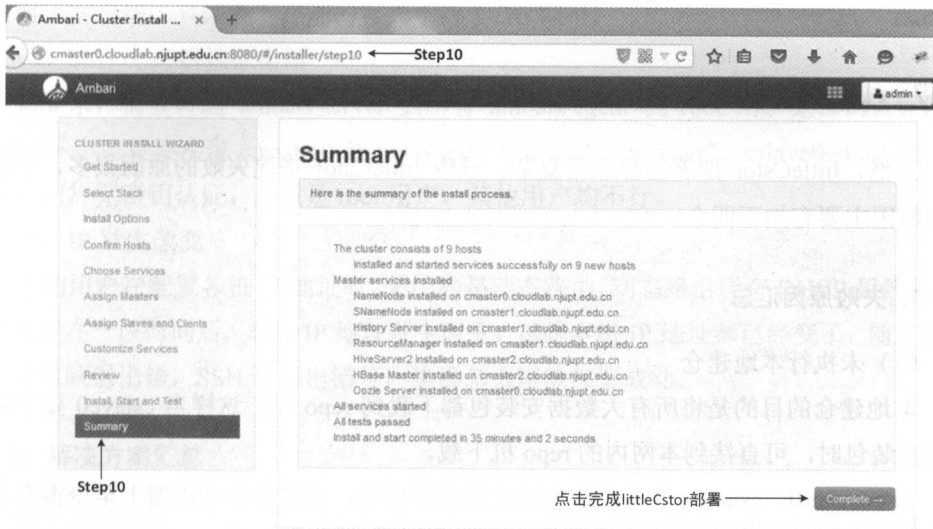


图 3-38 Summary

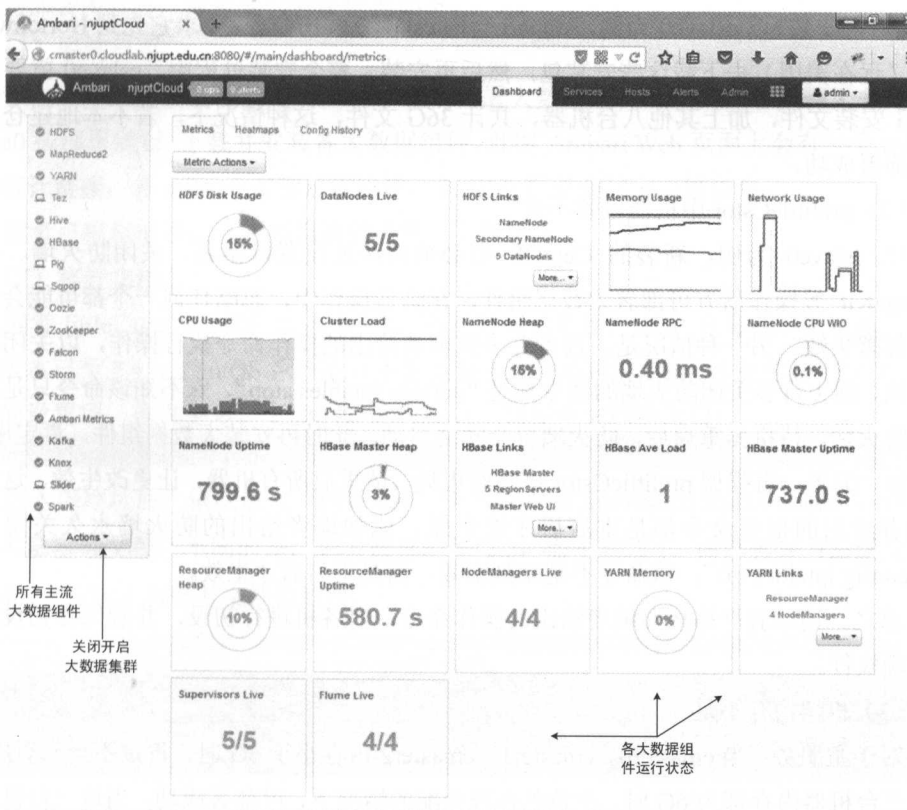


图 3-39 Dashboard

3.3.7 小结

显然, littleCstor 部署过程非常复杂, 导致 littleCstor 搭建失败的原因很多, 不过常见的原因主要有如下四个。

1. 失败原因汇总

(1) 未执行本地建仓

本地建仓的目的是将所有大数据安装包都下载到 repo 机, 这样当 cslave0 要下载大数据安装包时, 可直接到本网内的 repo 机下载。

当 cmaster0、cslave0 这些机器安装大数据组件时, 前提是这些机器上至少得有大数据安装包, 就像要在 brilliant (编者个人 PC, Win7 系统) 上安装 QQ, 就需要有 QQ.exe 一样, 当 Ambari 在这些机器上安装大数据组件时, 它实质上是指挥这些机器先下载安装包, 然后再安装。以 cslave0 为例, Ambari 会向 cslave0 发出命令, 要求它先到 Hortonworks 仓库 (远在美国) 去下载这些安装包, 然后再安装。整个安装过程中, cslave0 就要下载近 4G 安装文件, 加上其他八台机器, 共计 36G 文件, 这种情况下, 若不本地建仓, 不可能部署成功。

(2) prelittleCstor 中机器设置不对

以 cslave0 为例, 新装的 CentOS 机必须要经过设置机器名、关闭防火墙、关闭 PackageKit 等操作后方可部署大数据组件, 在这些操作中, 忽略任何一个都可能会导致后期部署失败。另一种情况是, 读者并未按编者给出的操作命令执行操作, 以关闭防火墙为例, 网上许多关闭防火墙的命令都是 “service iptables stop”, 殊不知该命令只是临时关闭防火墙, 待机器重启后, 防火墙也会随之启动, 此时再安装大数据组件, 肯定出错。再比如, 编者一再强调 prelittleCstor 搭建结束后, 请重启所有机器, 让更改生效, 这是因为编者给出的很多命令都是重启后才能生效, 比如编者给出的防火墙永久关闭命令 “chkconfig iptalbes off”, 该命令不是立刻生效, 而是重启后才生效。

总之, 请读者严格按照编者给出的操作命令, 对各机进行初设, 并确保各初设操作被正确执行。

(3) 机器内存不足

对于主服务: 当 cmaster0、cmaster1、cmaster2 内存小于 6G 时, 肯定不会部署成功; 当这三台机器内存都为 6G 时, 在角色合理分配的情况下, 可部署成功; 当这三台机器内存都大于 8G 时, 可放心大胆地部署。

对于从服务: 以 cslave0 为例, 当 cslave0 内存小于 4G 时, 基本不会部署成功; 当 cslave0 内存大于等于 4G 时, 可放心部署。

对于客户端：只须各客户端内存大于等于 2G 即可。

(4) SSH 无密钥认证出错

按要求，需要打通 ambari-server 到所有 ambari-agent 机 root 用户无密钥认证。在 littleCstor 中就是要实现 cmaster0 到 cmaster0、cmaster1、cmaster2、cslave0~4 和 iclient0 机 root 用户无密钥认证。注意是 root 用户，其他用户均不行。

(5) IP 发生改变

有的用户在配置各机 IP 地址时，采用的是动态路由，动态路由皆有 DHCP 租约问题，极有可能在一段时间后，本机 IP 地址发生改变，试想本机 IP 地址都已经变了，随之而来就是域名映射出错，SSH 认证出错等，集群怎么可能部署成功。

2. 解决方案汇总

除了上述原因，其他原因也会导致 littleCstor 部署失败，比如 cslave0 机上正在运行一客户程序，恰好该程序占用了大数据组件需要使用的端口，从而导致部署失败。当读者遇到部署困难时，最重要的解决依据如下。

(1) 安装日志

以 cslave0 为例，当使用 Ambari 自动部署大数据组件时，其部署步骤无非是要求 cslave0 按部署规划、下载并安装各大数据组件。此时 Ambari Web 页面上会有一个 cslave0 机的统计链接，若 cslave0 安装过程出错，该 Web 页面会显示出错信息。

读者可根据出错信息，利用搜索引擎，寻找答案。

(2) 官方文档

编者 littleCstor 中 HDP 版本为 2.2，随着时间的推移，HDP 自身在不断更新（截至当前版本已至 2.4），CentOS 系统也在不断更新。也许最新的 CentOS 在部署 HDP 时需要更少的设置项，也许更多。下面的地址为本书一切参考文档的入口：

<http://docs.hortonworks.com/>

进入该地址后，依次进入“HDP 2.4”→“Ambari Automated Install Guide”，该文档即为本章参考文档，不同的是，编者选择的是 2.2，而不是 2.4，请读者参考最新文档，图 3-40、图 3-41 为参考文档位置图。

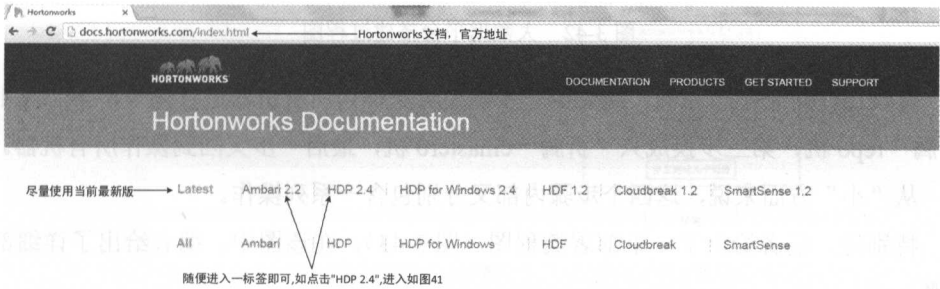


图 3-40 HDP 官方文档

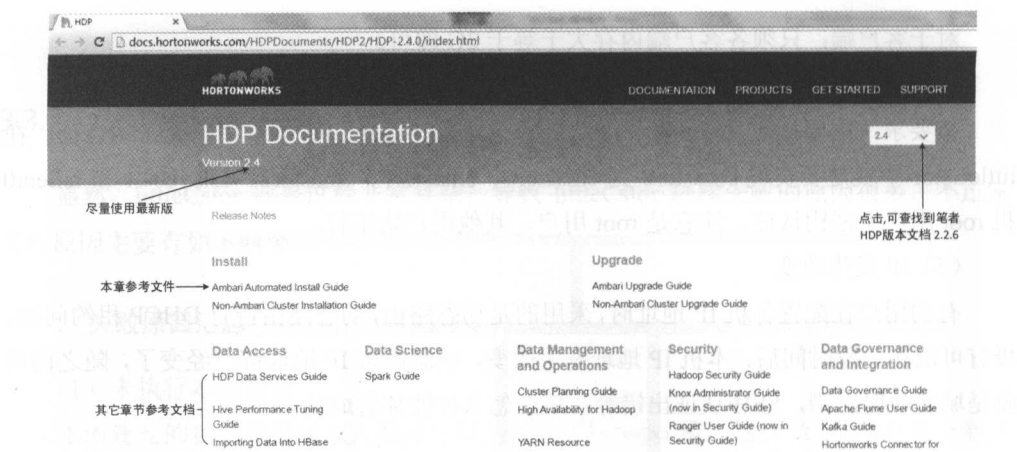


图 3-41 Ambari 自动部署大数据集群文档

编者参考的是“Hortonworks Data Platform 2.2”，若读者直接阅读英文，建议参阅最新文档（比如当前的“Hortonworks Data Platform 2.4”）。该文档下的“Ambari Automated Install”是一切操作的标准。

(3) 搜索引擎

遇到问题时，请根据 log 日志找出问题点，然后使用搜索引擎寻找该问题解决方案。搜索过程基本为：使用百度→使用 bing 英文模式→想办法使用谷歌。

(4) 专家指点

部署过程中，遇到 CentOS 问题可请教 Linux 高手，遇到大数据组件问题可请教专业机构（如南京云创）。

3. 部署步骤汇总

从“大”方面来说，除了部署规划外，littleCstor 搭建过程包含如下四步（图 3-42）。

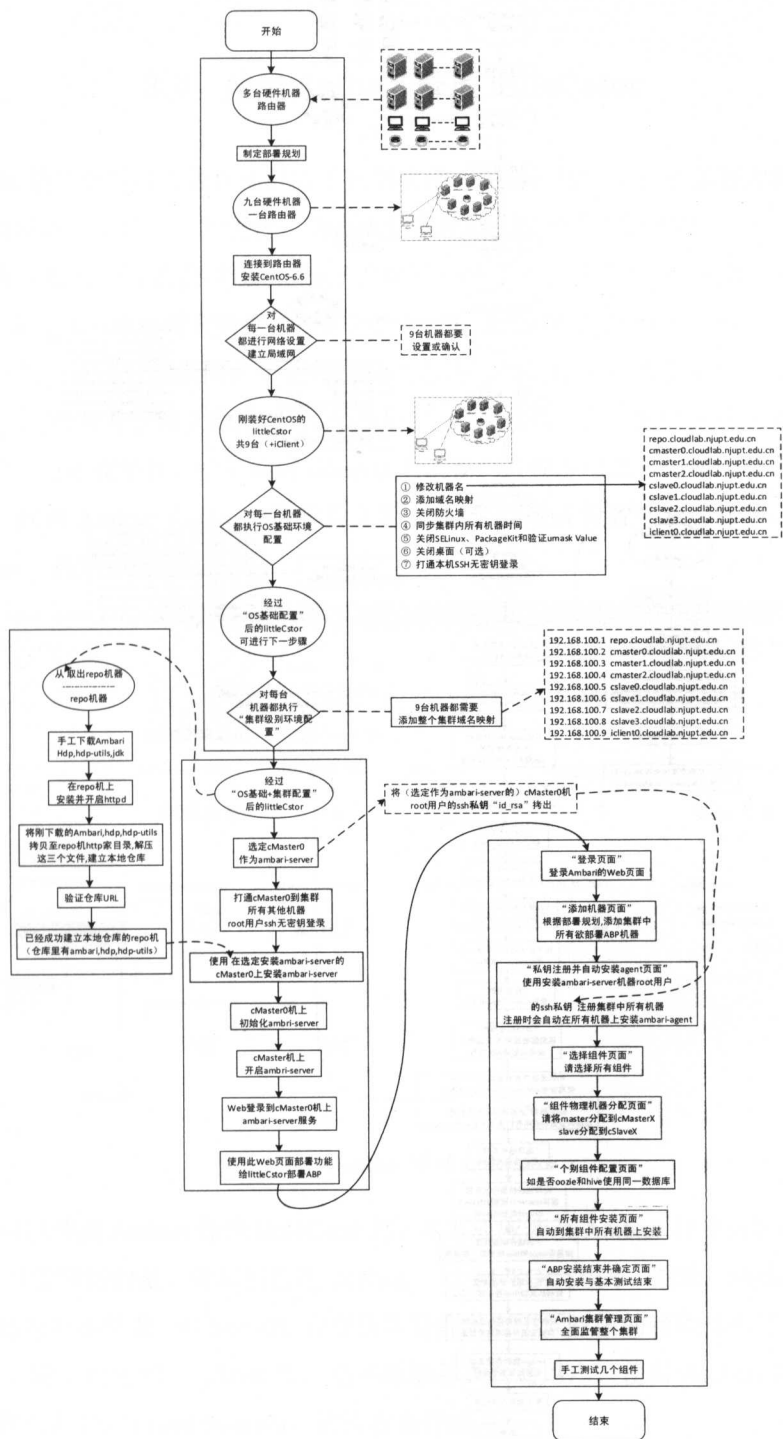


图 3-42 大数据平台搭建过程图

各步操作对象不尽相同，比如第一步需要对集群内所有机器进行操作，第二步则只“折腾”repo 机，第三步换成只“折腾”cmaster0 机，最后一步又回到操作所有机器。

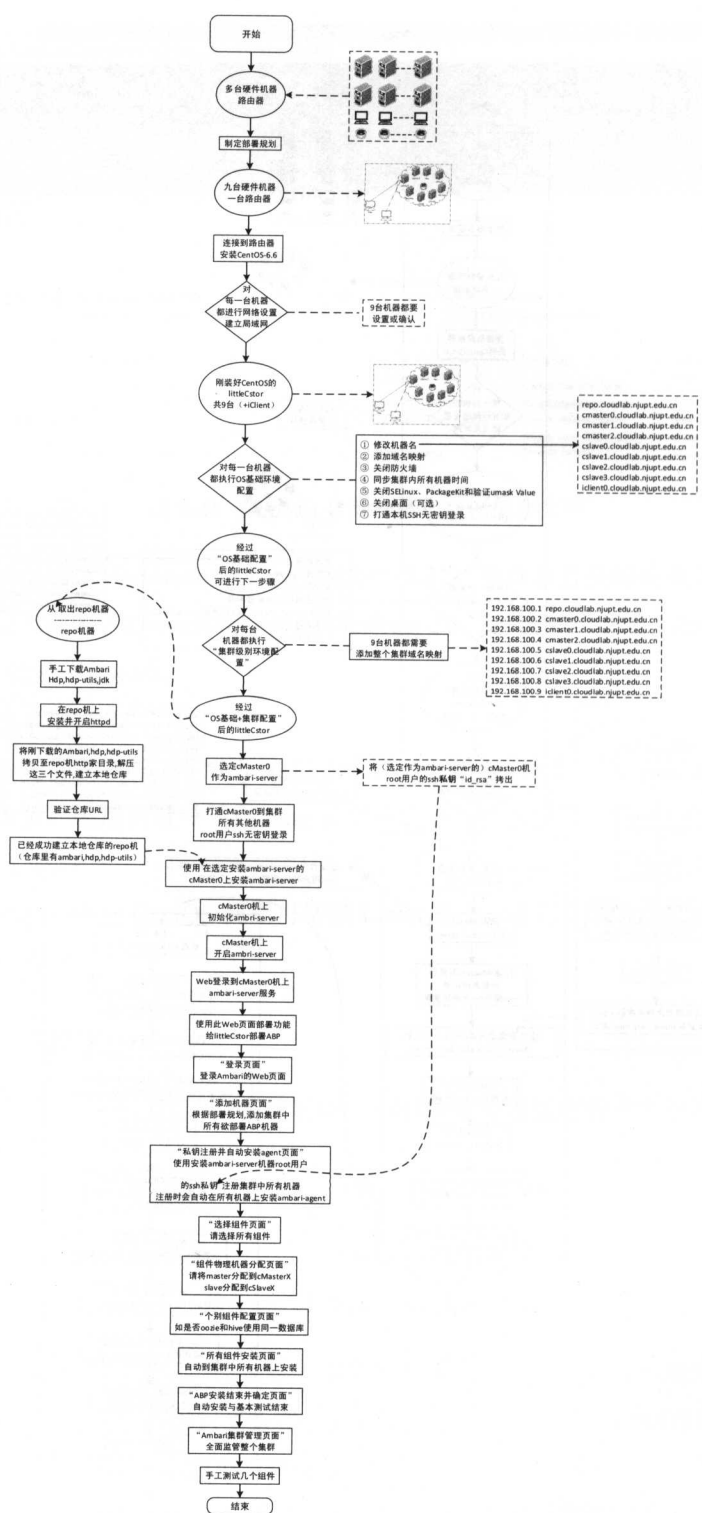
从“小”方面来说，这四个步骤内部又分别包含一系列操作。

特别地，编者给出了一个部署流程图（图 3-43），在该图中，编者给出了详细部署步骤。



(a)

图 3-43 littleCstor 部署流程图



(b)

图 3-43 littleCstor 部署流程图 (续)

3.4 使用 Ambari 管理 littleCstor

Ambari 是一个管理大数据集群的工具型软件，我们可用使用它来部署大数据组件、启动大数据组件、监管大数据组件、查看大数据组件、关闭大数据组件。

本章几乎所有内容都在讲述 Ambari 的部署功能，该功能自不必说；图 3-44 即显示了 Ambari 具有开启和关闭大数据集群的功能；这里所谓的查看是指当管理员想要“主动”查看某机或某组件运行状况时，可通过链接“点进”该组件；和查看不同，监管指的是 Ambari 会自动监督并管理大数据组件，一旦发生组件故障，会以邮件方式立刻通知管理员。比如管理员正在午休，恰好此时 cslave0 上某客户进程占用太多内存而导致某大数据组件崩溃，此时 Ambari 会自动向管理员邮箱（通过 Ambari 设置）发送邮件，请读者登录 littleCstor，熟悉各种管理工具。

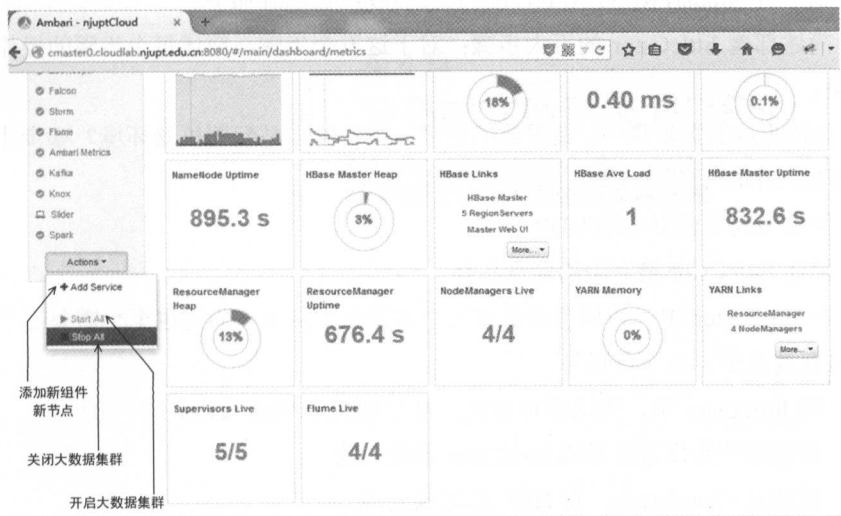


图 3-44 Ambari 管理 littleCstor

除了可以使用 Ambari 管理 littleCstor 外，本节还应涉及 Ambari 自身的管理，所谓的 Ambari 自身管理指的是，管理员管理 Ambari。以 MySQL 为例，诚然，MySQL 用来管理存储在其内的各种表。但 MySQL 自身还需管理具有不同权限的管理员账号，类似，在 Ambari 中，除了超级用户 admin 外，还可添加各类普通用户。使用 Ambari 用户管理功能，可依次点击 Dashboard→admin，请读者自行练习。

3.5 小结

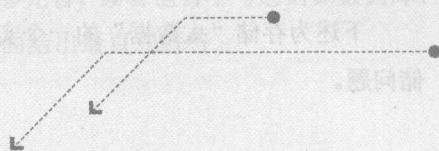
至此, 本章结束。本章首先讲述了常见的大数据管理工具, 然后重点讲解了使用 Ambari 部署大数据集群, 最后, 编者给出了使用 Ambari 管理大数据集群的常见操作。

习 题

1. 既然可以采用手工方式部署大数据组件, 为什么还要借助 Ambari?
2. 简述 Ambari 功能作用及其体系架构。
3. 为何部署 Ambari 时只需要部署 AmbariServer?
4. 试通过 Ambari REST APIs, 编写代码获取 Ambari 服务。
5. 简述部署 littleCstor 的五大步骤; 对于这五个步骤, 简述每个步骤的操作前提和操作结果。
6. 你认为这五大步骤中, 最重要的是哪一步? 哪步是配置机器环境? 哪步才是使用 Ambari 部署 HDP?
7. 简述为何要建立本地 HDP 仓库。
8. 简述 littleCstor 最完美的物理拓扑。
9. 在 littleCstor 中, 为何至少需要三个主节点? 从节点为何也至少需要三个? 主节点最低内存是多少? 从节点呢?
10. 在 littleCstor 中, 采用何种方式, 可尽量减少 Web 攻击?
11. 简述哪些操作肯定导致 littleCstor 部署失败。
12. 试通过 Cloudbreak, 将 HDP 部署到云端。

参考文献

- [1] http://wikibon.org/wiki/v/The_Hadoop_Wars:_Cloudera_and_Hortonworks%E2%80%99_Death_Match_for_Mindshare
- [2] <http://docs.hortonworks.com/index.html>
- [3] https://en.wikipedia.org/wiki/Secure_Shell
- [4] https://en.wikipedia.org/wiki/Secure_copy



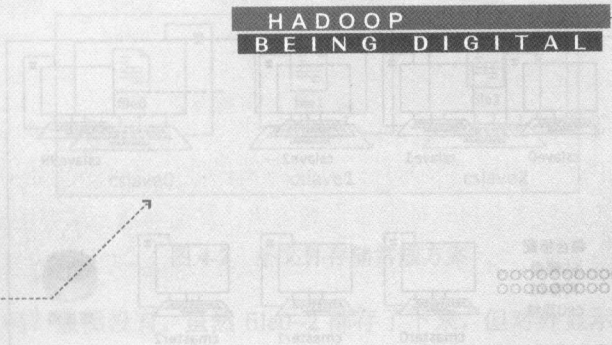
第4章 分布式文件系统 HDFS

第4章

分布式文件系统 HDFS

HADOOP

BEING DIGITAL



随着移动互联网的快速发展，特别是以博客、社交网络为代表的新型信息发布系统在移动终端的成功应用，数据正以前所未有的速度在不断地增长着。大数据时代已经来临，而大数据面临的首要问题就是存储。本章首先通过实例引出分布式存储；接着重点讲解当前主流大数据存储工具—HDFS；最后以实战方式讲述 HDFS 应用实例。

4.1 分布式存储引例

下述为存储“大数据”的一个实例，请读者在此场景下分析、讨论并解决大数据存储问题。

4.1.1 问题描述

例 1 现有一些配置完全相同的机器 `cmaster0`、`cmaster1`、`cmaster2`、`cslave0`~`cslaveN`，已知每台机器都是 1 个双核 CPU，5GB 硬盘，4G 内存（图 4-1）。请按下述要求回答问题。

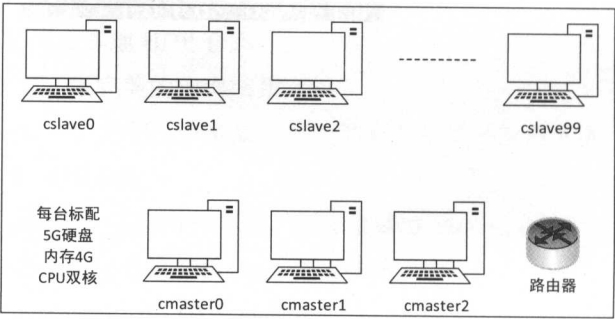


图 4-1 场景硬件配置图

1. 第一类问题：存储

问题①小文件统一存储：现有三个大小都是 3GB 的文件 `file0`、`file1` 和 `file2`，要求在任选机器的情况下，将这三个文件存入机器，并且对外显示时（逻辑上）这三个文件存于同一个存储空间。

问题②大文件独立存储：现有一个大小为 6G 的新文件 `file9`，要求存入机器后对外显示时，逻辑上依旧为一个完整文件。

2. 第二类问题：计算

问题③：在问题①下，统计 file0 和 file1 这两个文件里每个单词出现的次数。

问题④：在问题②下，统计 file9 中，每个单词出现的次数。

3. 第三类问题：可靠性

问题⑤存储可靠性：假设用于解决上述问题的机器宕机了，如何保证数据不丢失？

问题⑥计算可靠性：假如 cs slave0 正在计算 file0，但在计算过程中 cs slave0 突然宕机了，如何保证计算任务继续进行？

为求简单明了，上述场景与问题的描述可能不够完善，读者也暂不考虑诸如数据库、压缩存储、NFS 等方案，把思路放在分布式上，下面给出最直观解答。

4.1.2 常规解决方案

1. 问题①小文件统一存储

解 取三台机器 cs slave0、cs slave1 和 cs slave2，将 file0 存于 cs slave0、file1 存于 cs slave1，同理 cs slave2 存储 file2（图 4-2）。

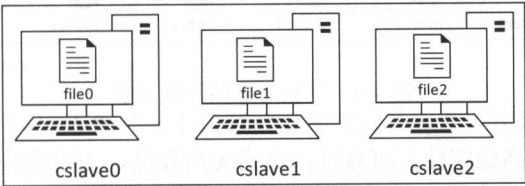


图 4-2 小文件存储常规方案

问题解决了吗？显然没有，虽然 file0~2 都存了下来，但对外显示时它们并非存于同一个存储空间，似乎除非将三块硬盘连接到一起，才能解决这个问题，可明显这三块硬盘属于三台不同机器。

2. 问题②大文件独立存储

解 为保证单台机器能够存下 file9 这个“大”文件，首先，将此文件拆成两个大小相等的文件 file9-a 和 file9-b，接着将 file9-a 存于 cs slave0 上，file9-b 存于 cs slave1 上(图 4-3)。

由于硬盘只有 5G，任一单台机器都存不下 file9 这个“大”文件，通过拆分，我们成功地实现了文件存储，可由于拆分后的文件存于两个硬盘，对外显示时它们实际上已经是两个不同文件，分属于两个不同的存储空间，问题②也没有解决。和问题①类似，如果能将三台机器存储空间连成一片，此问题好像很容易解决。

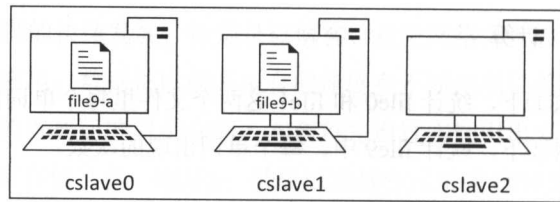


图 4-3 大文件存储常规方案

3. 问题③存储可靠性

解 为了防止数据丢失，办法之一是将普通机器换成最贵、最好、最稳定的服务器，并为服务器配置 RAID（磁盘阵列），最后将这些服务器安置在最好的机房里（图 4-4）。

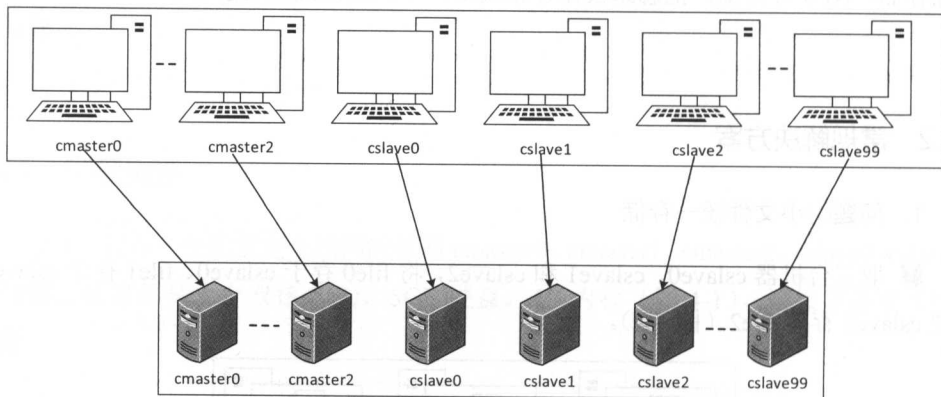


图 4-4 存储可靠性常规方案

硬件自身要提供高稳定性、可靠性，这点无可厚非，可现实中决不能仅依赖硬件可靠性，比如机房所在地发生台风、地震等自然灾害，甚至普通火灾都可能导致数据丢失。那如何提高存储可靠性呢，分布式环境下采用的办法很简单——冗余。

综上所述，常规思路下，物理上，似乎只有对每台机器进行扩容，才能解决问题，可如果例题中的硬盘配置不是 5G 而是 5T，一方面，不断扩容不仅降低了机器性能，也总会达到单机存储空间上限，另一方面，无论怎么扩容都会遇到存不下的情况。诚然，单机应具有足够的性能以解决大部分问题，可我们不当依赖单台机器解决所有问题。

逻辑上，如果能在 cslave0、cslave1 和 cslave2 上部署一层软件，并依靠此软件将这三台机器的三个独立存储空间连成一个完整空间，那么问题①、②都将得到解决。由于连接后的存储空间容量增大三倍，（逻辑上）存储区域完全统一，此存储系统可完美解决小文件统一存储和大文件独立存储问题，这个方案也正是下一节的分布式解决方案。

4.1.3 分布式解决方案

上述方案并没有真正解决问题，下面介绍的分布式方案也是 HDFS 的架构思路，读者须仔细研读，重点理解其架构思想，至于有些不好理解的地方，暂不必追究，后面章节将深入讲解。

1. 分布式存储

几乎所有的分布式系统都采用 master/slave 架构^[2]，其中 master 一般会负责指挥和管理，非常智能。而 slave 则正好相反，一般都是机械地执行 master 发来的命令，没有一点“主动性”。此处将构建的 DFS 也采用这种架构，下面以例题形式，给出构建过程。不过，在构建之前，需引入文件块概念，如下：

将文件按固定大小分成多个部分，每个部分称为一个文件块。

比如当假定块大小为 3G，则大小为 3G、6G 和 7G 的三个文件分别可划分成 1 个块、2 个块和 3 个块。这是因为，由于块大小为 3G，故 3G 文件会占用一个完整块，同理 6G 文件占用两个完整块。由于“ $7/3=2$ 余 1”，故 7G 文件会占用 3 个块，不过这三个块中只有前两个是完整块，第三个块只使用了 1G，不会占用整个块空间。

1) 分布式存储架构

对于第一类存储问题，若能将每台机器的存储空间（硬盘）以某种方式连接到一起，则问题迎刃而解。将这些机器成功连接后，首先就要面临如下两个问题：

- 谁来存储实体数据；
- 谁来管理统一空间。

为此，可采用客户-服务器模式（客户端主动连接，服务器被动接受，并且客户端取得服务方式相同），按如下三步，构建图 4-5 中的分布式存储集群。

Step1 首先取机器 cslave0、cslave1、cslave2 和 cmaster0。

Step2 将这些机器角色分为 master 和 slave 两类，这里不妨设定 cmaster0 为 store master，cslave0、cslave1 和 cslave2 为 store slave。

Step3 规定 store master 不存储数据，统一管理所有 store slave 存储空间，store slave 作为实际存储节点，逻辑上其存储空间交由 store master 统一管理，物理上 store slave 用来存储真实数据。

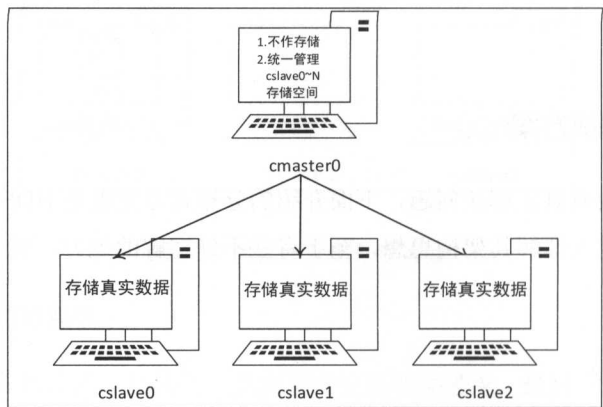


图 4-5 分布式存储架构图 1

经过上述方式构建后的集群，对内由于采用客户-服务器模式，只要保证 store master 正常工作，我们很容易随意添加 store slave，硬盘存储空间无限大。对外，整个集群就像是一台机器、一片云，硬盘显示为统一存储空间，文件接口统一（图 4-6）。

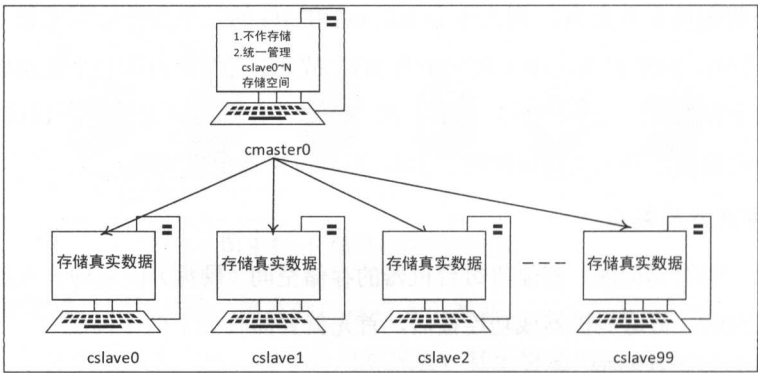


图 4-6 分布式存储架构图 2

称此新构建的文件系统为分布式文件系统（Distributed File System，DFS），由于存储空间统一且无限大，这个 DFS 可完美地解决问题①、②。Hadoop 分布式文件系统（Hadoop DFS，HDFS）的架构思想和上述过程类似。

2) 分布式文件系统中各机角色定位

显然，该系统最终是给用户使用的，故系统中肯定有客户端（图 4-7），不过应当注意的是客户端实际上根本不知道还存在着 slave 机器，它只知道 cmaster0 的网络位置和相关存储协议，当发生具体存储时，cmaster0 才将对应的 slave 机位置和权限信息发往 iclient0。

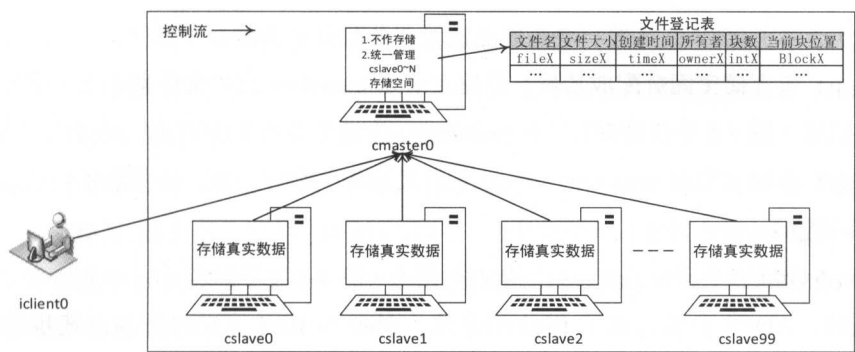


图 4-7 包含客户端的 DFS 架构图

除了 client 角色外，从大角度来说存在集群和客户端两个角色，而集群里又有主从节点之分；功能方面，不同定位的机器，具有完全不同的功能，具体可参见表 4-1。

表 4-1 功能定位表

角 色	定 位	机 器	功 能
集 群	存储主节点	cmaster0	1) 不存储具体文件
			2) 统一管理所有 slave 机存储空间
			3) 维护一张“文件登记表”
	存储从节点	cslavex	1) 存储某块具体文件 2) 实际响应客户端读写文件操作
客户端	客户机	iclient0	集群使用者

不妨称此分布式存储系统为 ADFS (Allen Distributed File System, Allen 为编者英文名)，尽管 ADFS 的相关协议不够细致，但它能够清晰地阐述 HDFS (Hadoop DFS) 的层次架构。

3) 小文件存储

在 ADFS 中，由于 cmaster0 已将所有 slave 存储空间连到一起，ADFS 存储空间无限大，存储接口统一。下面我们以 Grid 用户上传 file2 为例，简述 ADFS 存储过程，其他两个用户上传文件时操作相同。

Step1 Grid 用户客户端根据预先设定的存储块大小，计算文件块个数，本例中块大小为 3G，故 file2 占用了完整块 (图 4-8 中①)。

Step2 Grid 用户主动连接存储主节点 cmaster0，告知 cmaster0 本次要存入文件的相关信息 (图 4-8 中②)，当然除了最基本的文件名、文件大小、所有者等信息外，还应包含此文件应当占用的块数量 (Step1 计算所得)。

Step3 存储主节点 cmaster0 遍历所有 slave 存储空间 (实际上是遍历文件存储表，图 4-8 中步骤③)，找出一片较为空闲的存储区，接着将这片存储区分配给 file2，本例中

cmaster0 选中了 cs slave2 上的一个完整存储块并分配给了 file2。

Step4 待存储空间分配成功后, 存储主节点 cmaster0 在“文件登记表”中登记此文件相关信息 (图 4-8 中步骤④), 图中 cmaster0 登记了 file2 的文件名、大小等信息。

Step5 存储主节点 cmaster0 向 cs slave2 发出相关存储命令, 让其准备接收数据 (图 4-8 中步骤⑤)。

Step6 Grid 客户端向 cs slave2 上传文件 file2 (图 4-8 中步骤⑥)

至此, ADFS 已经完成了 file2 的存储。file0 与 file1 存储过程与上述步骤类似, 从图 4-8 中我们能够看出, cs slave0 存 file0, cs slave1 存 file1, cs slave2 存储 file2, 这似乎和“常规解决方案”中没有任何区别, 但实际上当前所有 cs slave 的存储空间已经由 cmaster0 统一管理, 存储空间只有一个, 不存在其他存储区域。

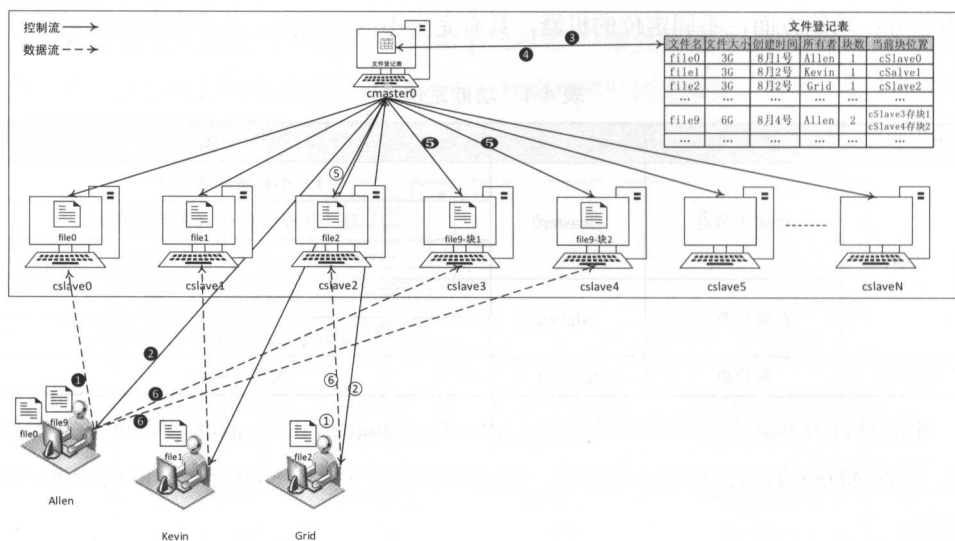


图 4-8 分布式存储架构图

读者要注意的是, 在传输过程中 file2 实际上并未经过 cmaster0, 而是根据 cmaster0 的授权, Grid 客户直接将 file2 传到了 cs slave2。试想, 如果上传的所有文件都要经过 cmaster0, 那 cmaster0 必然成为整个 DFS 最大的瓶颈, 几个文件同时传输就能让 cmaster0 因网络瘫痪而宕机, 届时整个 ADFS 也将不复存在。

4) 大文件存储

由于此 ADFS 存储空间无限大, 完全满足 file9, 可直接将 file9 存入此 DFS。实际操作时, 大体应遵循如下步骤。

Step1 Allen 客户端根据预先设定的存储块大小, 计算文件块个数 (图 4-8 中①), 本例中块大小为 3G, 故 file9 将会占用两个完整块。

Step2 Grid 客户端主动连接存储主节点 `cmaster0`, 告知 `cmaster0` 本次要存入文件的相关信息 (图 4-8 中 ②), 当然除了最基本的文件名、文件大小、所有者等基本信息外, 还应包含此文件应当占用的块数量 (Step1 计算所得)。

Step3 存储主节点 `cmaster0` 遍历所有 `slave` 存储空间 (实际上是遍历文件存储表, 图 4-8 中步骤 ③), 找出两片较为空闲的存储区, 接着将两片存储区分配给 `file9`, 本例中 `cmaster0` 分别选中 `cslave2` 和 `cslave3` 上的两个完整存储块并指定 `cslave2` 存储“file9-块 1”、`cslave3` 存储“file9-块 2”。

Step4 存储主节点 `cmaster0` 确定此次分配成功后, 在“文件登记表”中登记此文件相关信息 (图 4-8 中步骤 ④), 本题中 `cmaster0` 登记了 `file9` 的文件名、块数量和每个块存储的具体位置等信息。

Step5 存储主节点 `cmaster0` 向 `cslave3` 和 `cslave4` 发出相关存储命令, 让其准备接收数据 (图 4-8 中步骤 ⑤)。

Step6 Allen 客户端先向 `cslave3` 上传“file9-块 1”, 再向 `cslave4` 上传“file9-块 2” (图 4-8 中步骤 ⑥)。

从上述步骤可以看出, 大文件存储过程和小文件大致相同, 不同的是大文件一般包含多个块, 文件登记表里对每个块的具体位置都要进行记录。此外虽然 `file9` 在存储时也被分割成了两个块, 但由于整个集群已经是一个完整的文件系统, 用户并不知道文件已经划分成了块。当然, 此处 `file9` 上传过程中也不会经过 `cmaster0`。

2. 可靠的分布式存储

解决可靠性最直接的方式就是对数据进行备份, 可数据备份会造成多版本问题, 谷歌采用的“冗余”机制其实是“备份”的简化版, 它只会保存最新数据的一个或多个副本, 完全避开了多版本问题, 不过更为精妙的是当冗余机制和 DFS 结合时, 在分布式计算端将会大放异彩, 这些我们将会在第六章具体介绍, 本节以分布式存储为主, 下面以实题形式讲解 DFS 使用“冗余”机制来保障存储可靠性。

以块为单位, 对于同一个块, 确保整个集群内这个块有一个或多个副本。此操作应当由存储主节点自动维护, 即一旦存储主节点发现集群内某个块只有自己本身, 则立即调用块分布算法, 复制此块, 并存于其他机器。

下面以图 4-9 中的两副本为例, 说明如何使用冗余方案解决可靠性问题, 对于同一个块, `cmaster0` 会确保集群内始终有两个, 这样即使某台机器宕机, 集群依旧不会丢失数据。比如由于 CPU 故障, `cslave1` 宕机, 导致存储在 `cslave1` 上的 `fileX-块 2` 丢失, 但由于 `cslave99` 上同样存储着 `fileX-块 2`, 故对于整个集群来说, 数据并未丢失。不过此时 `cmaster0` 会感知到整个集群内只有一个“fileX-块 2”, 它会再次启动拷贝操作, 拷贝一个

“fileX-块 2”到集群的另外一台空闲机。

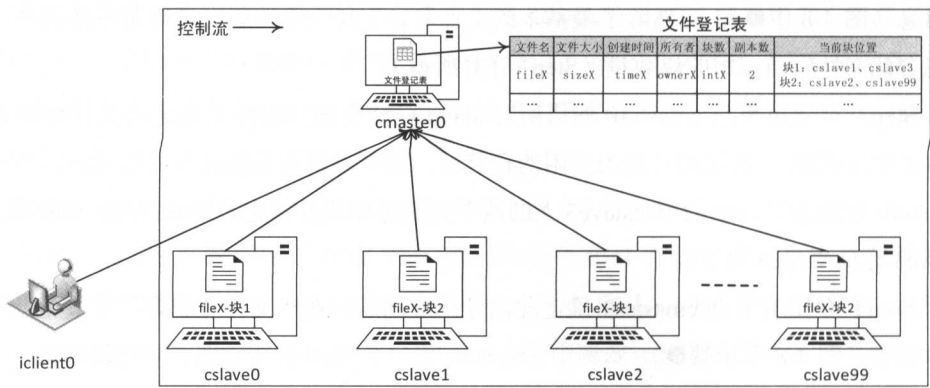


图 4-9 冗余可靠性示例图

1) 存储步骤

如下的 19 个步骤将描述 ADFS 针对 file9 的存储过程，请读者参考图 4-10，理解集群内块复制过程。

- Step1** iclient0 (Allen) 根据存储协议划分 file9。
- Step2** iclient0 (Allen) 主动连接 cmaster0。
- Step3** cmaster0 遍历文件登记表，寻找两个空闲块并试图分配这两个块。
- Step4** 存储空间分配成功后，cmaster0 在文件登记表上登记 file9。
- Step5** cmaster0 指示 cs slave1 和 cs slave3 接收数据。
- Step6** iclient0 上传“file9-块 1”至 cs slave1。
- Step7** cs slave1 接收完“file9-块 1”后向 cmaster0 汇报接收成功。
- Step8** cmaster0 指示 cs slave0 准备接受“file9-块 1”。
- Step9** 根据 cmaster0 指示，cs slave1 将“file9-块 1”发送到 cs slave0。
- Step10** cs slave0 向 cmaster0 汇报“file9-块 1”接收成功。
- Step11** iclient0 上传“file9-块 2”至 cs slave3。
- Step12** cs slave3 接收完“file9-块 2”后向 cmaster0 汇报接收成功。
- Step13** cmaster0 向 iclient0 说明文件存储成功。
- Step14** cmaster0 指示 cs slave5 准备接受“file9-块 2”。
- Step15** 根据 cmaster0 指示，cs slave3 主动将“file9-块 2”发送到 cs slave5。
- Step16** cs slave5 向 cmaster0 汇报“file9-块 2”接收成功。

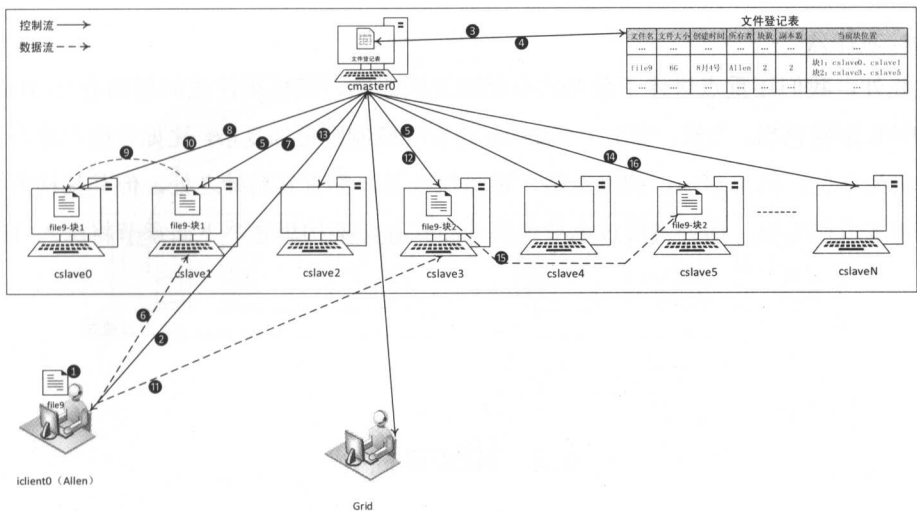


图 4-10 ADFS 可靠性存储

从示例图可以看出，Step8、9、10 和 Step14、15、16 才是集群内块的复制过程，并且这个过程和 iclient0 无关，cmaster0 在集群内部全程指挥该过程，iclient0 端并不知道（不需要，也不应当知道）此动作。从 Step13 也可以看出，当上传完 file9 后，cmaster13 立即向 iclient0 端返回文件上传成功，此时集群内部极有可能还在复制数据。

2) 优劣分析

对于第二问，在使用冗余机制的 DFS 集群内，每台服务器都不需要安装 RAID^[3]（磁盘阵列），从单台机读写角度上来说，装有 RAID 的单机读取数据的速度比普通机器快许多，但事实上 DFS 不仅仅是硬盘的扩展，在存取数据时，参与者不仅仅是数据所在机硬盘，还包含所在机 CPU、内存和网络，故从整个集群角度来说，DFS 的数据存取速度是 RAID 的很多倍。使用冗余机制的 DFS 优点明显，首先能够防止数据丢失，确保数据安全；其次，它能够大大加快数据存取速率，此外，在分布式计算章节，读者还将看到它的更大作用。

冗余机制的 DFS 最大的缺点就是存储空间消耗太快，试想如果 1T 数据有三个副本，本来这 1T 数据普通环境下只占用 1T 空间，可在冗余机制的 DFS 里却需要使用 3T 空间来存储，即使对于 DFS 这种海量存储系统也相当耗费资源。此时我们可采用二进制存储、压缩存储、归档存储、优化块大小、调节块副本数等机制（或多个机制组合）对 DFS 进行调优，根据编者经验，经过上述优化机制组后的存储空间可降至优化前一半，比如优化前 1T 数据可优化成 500G 左右。

至此，分布式存储引例结束，本节通过实际场景，回答了分布式存储最基本问题：

- 是什么

- 为什么

此外,我们还重点讲述了分布式存储的架构思路,读者须注意此思路也是 Hadoop 的 HDFS 架构思想。当然,现实中 Hadoop 的实现机制则更加复杂,比如它的存储与计算都以块为单位、机架感知、调度策略、推测执行等,都有其精妙之处,但其架构的基本思路和本节很类似,读者可通过本节理解分布式存储架构思想,下面章节将深入 HDFS 架构。

4.2 HDFS 简介

参照 Google 论文 GFS^[1], Apache 实现了 Hadoop 版的分布式文件系统 HDFS,由于 HDFS 自身的成熟稳定,加之用户众多, HDFS 已经成为当前分布式存储的事实标准。

作为 Hadoop 的核心技术之一, HDFS (Hadoop Distributed File System) 为大数据平台其他所有组件提供了最基本的存储功能。其所具有的高容错、高可靠、高可扩展、高获得性、高吞吐率等特征为大数据存储和处理提供了强大的底层存储架构,可以说它是一切大数据平台的基础。

4.2.1 HDFS 逻辑架构

HDFS 是一个主/从 (master/slave) 体系结构,如图 4-11 所示。从最终用户的角度来看,它就像传统的文件系统一样,可以通过目录路径对文件执行 CRUD (Create、Read、Update 和 Delete) 操作。但由于分布式存储的性质, HDFS 集群拥有一个 NameNode 和一些 DataNodes。NameNode 管理文件系统的元数据, DataNode 存储实际的数据。客户端通过同 NameNode 和 DataNodes 的交互访问文件系统。客户端联系 NameNode 以获取文件的元数据,而真正的文件 I/O 操作是直接和 DataNode 进行交互的。

HDFS 开放文件系统的命名空间以便让用户以文件的形式存储数据。HDFS 的数据都是“一次写入、多次读取”,典型的块大小是 128MB。NameNode 执行文件系统的命名空间操作,比如打开、关闭、重命名文件或目录,决定数据块到 DataNode 的映射等。DataNode 负责处理客户的读写请求,依照 NameNode 的命令,执行数据块的创建、复制、删除等工作。例如客户端要访问一个文件,首先,客户端从 NameNode 中获得组成该文件的数据块位置列表,即知道数据块被存储在哪些 DataNode 上;然后,客户端直接从 DataNode 上读取文件数据。此过程中, NameNode 不参与文件的传输。

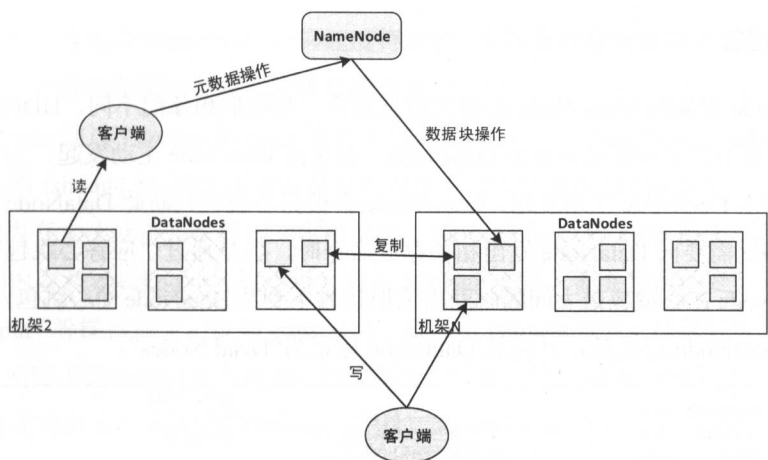


图 4-11 HDFS 的结构示意图

HDFS 典型的部署是在一个专门的机器上运行 NameNode，集群中的其他机器各自运行一个 DataNode；也可以在运行 NameNode 的机器上同时运行 DataNode，或者一台机器上运行多个 DataNode。一个集群只有一个 NameNode 的设计大大简化了系统架构。

NameNode 使用事务日志（EditLog）来记录 HDFS 元数据的变化，使用映象文件（FsImage）存储文件系统的命名空间，包含文件的映射、文件的属性信息等。事务日志和映象文件都存储在 NameNode 的本地文件系统中。NameNode 启动时，从磁盘中读取映象文件和事务日志，把事务日志的事务都应用到内存中的映象文件上，然后将新的元数据刷新到本地磁盘的新的映象文件中，这样可以截去旧的事务日志，这个过程称为检查点（Checkpoint）。HDFS 还设有 Secondary NameNode 节点，它辅助 NameNode 处理映象文件和事务日志。NameNode 启动的时候合并映象文件和事务日志，而 Secondary NameNode 会周期地从 NameNode 上复制映象文件和事务日志到临时目录，合并生成新的映象文件后再重新上传到 NameNode，NameNode 更新映象文件并清理事务日志，使得事务日志的大小始终控制在可配置的限度下。

HDFS 架构图参见图 4-12，再深入 HDFS 架构之前，我们须知道如下几个基本概念。

1. 块

块即为数据的一个集合，HDFS 里按块来存储数据，当前默认块大小为 128MB，存储时 Client 须按客户端协议将文件划分为一系列块，NameNode 存储文件句柄信息，DataNode 存储具体数据块，为确保数据可靠性、提高读写性能 NameNode 会尽量确保每个数据块均匀地分散存储于不同的 DataNode 中。

2. 心跳包

DataNode 定期向 NameNode 汇报的信息集合。和我们想象的不同，HDFS 里主服务 NameNode 并不会主动连接从服务 DataNode，而是由 DataNode 主动发起。正常运行时，每次通信都由 DataNode 主动发起，NameNode 会根据心跳包判断此 DataNode 当前状态，当 NameNode 需要向 DataNode 发出相关存取命令时，也是通过“应答心跳包”传送的。如果 NameNode 在心跳包最大间隔时间内依旧接收不到某 DataNode 的心跳包，NameNode 会认为此 DataNode 已失效，并将此 DataNode 标记为“Dead Nodes”。

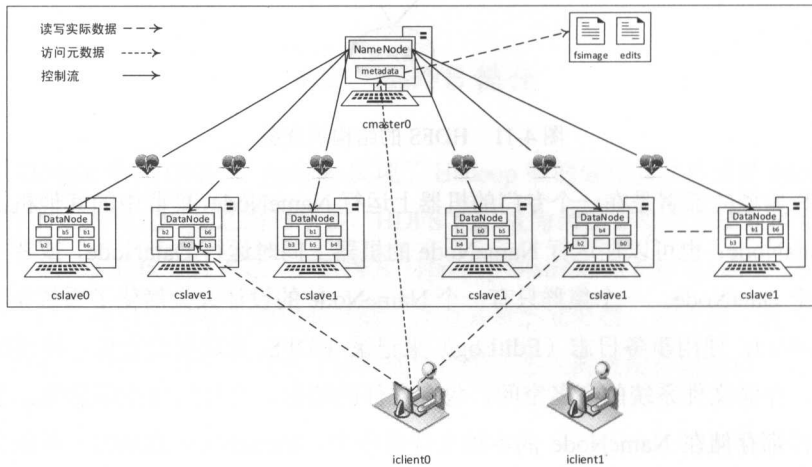


图 4-12 HDFS 架构图

HDFS 主要包括 NameNode（存储主节点），DataNode（存储从节点）和 Client（客户端）这三个实体，其中 NameNode 和 DataNode 为活动主体，Client 为非活动主体，具体讲解如下：

3. NameNode

HDFS 主进程名，整个集群中只有一个，通常，我们称运行 NameNode 进程的主机为存储主节点，作为守护进程，主节点须长期运行此进程，可以说 NameNode 开启即是集群的开启，NameNode 停止就是集群的停止。

从功能上看，NameNode 主要具有如下三大功能。

（1）管理 HDFS 命令空间

NameNode 通过 metadata 来管理整个文件系统。在 NameNode 运行时，它会在内存中维护整个 HDFS 最重要的数据 metadata（我们称之为 HDFS 元数据）。在设计之初，设计者并没有使用诸如 Derby、PostgreSQL 等第三方数据库来存储元数据，而是特意设计了一个非常紧凑的数据结构（metadata），并让 NameNode 在内存中直接操作，这个设计带

来了两大好处，首先是 NameNode 应答速度特别快，其次是存储容量巨大，仅 4G 内存的 NameNode 节点就能存储巨大的文件或目录索引。

NameNode 通过 fsimage 和 edits 来持久化 metadata。当 NameNode 启动时，它会将存储在本地的 fsimage 和 edits 中的内容写入内存中已实例化的 metadata 中；当内存中的 metadata 的内存大小或操作数超过一定阈值时，NameNode 会将 edits 操作写入硬盘 edits 文件，镜像内容和 checkpoint（检验点）写入 fsimage 文件。需要指出的是，对 HDFS 的“mkdir”、“put”等操作都会记录到 edits 文件，而不是 fsimage 文件，这极有可能造成 edits 本身过大，实际上此处设计极为精妙，为提高读写性能，防止 edits 文件过大，NameNode 采用轮询方式写出 edits。在 NameNode 关闭时，NameNode 在停止自身之前会将主要信息再次写入 edits 和 fsimage。

从 NameNode 角度来看，metadata 可靠性较低，一旦 edits 或 fsimage 丢失（比如本地硬盘故障），整个文件系统就会丢失，设计者采用 metadata 多处存储、NFS（网络文件系统）、SecondNameNode、Checkpoint Node、Backup Node 等机制来确保 metadata 性能（一致性、实时性、紧凑性、可靠性），此处过于复杂，我们不再讲解。

从 HDFS 角度来看，NameNode 本身就不是安全的，存在单点故障，在 NameNode 内容结束处，我们将讲解设计者如何解决此问题。

（2）协调整个 HDFS 存取

NameNode 负责集群所有存储任务，是 HDFS 所有决策的仲裁中心。从某种意义上说，NameNode 被设计得非常智能，而 DataNode 则毫无智能可言，它实际上就是一个命令执行者，只能被动地读写数据块，它甚至都不知道文件的存在。

（3）应答客户端存取请求

假如集群包含一万个 DataNode，那从 Client 角度来说，它不想知道这些 DataNode 是谁（难以维护 IP 和协议列表）；也不应当让 Client 知道集群内部拓扑（这样集群很容易受到恶意用户的攻击）。故从 Client 角度来说，它只知道 NameNode 的通信地址和存取协议，每次存取时，都是 Client 主动连接 NameNode，而当 Client 想要具体存取某块数据时，NameNode 会把对应机器（存储此数据块，且物理上离此 Client 最近）的协议信息发往 Client，让 Client 自己到该 DataNode 上存取。这样的设计能够最大程度减轻 NameNode 的负担。

正如 metadata 存在可靠性问题一样，从集群角度来说，NameNode 本身就是个单点故障，一旦 NameNode 宕机，整个集群服务都随之崩溃，此时唯一办法就是重启 NameNode。我们可以确定，无论多么稳定的单机（运行 NameNode）都无法确保 100% 可靠性，故为了确保 HDFS 提供 24 小时 365 天不间断服务，就必须对 NameNode 机进行双机甚至是多机热备。目前最常用的三大方案为 High Availability With QJM、High

Availability With NFS 和 Federation, 在这几个方案中, High Availability With QJM 效果最佳, 由于需要 ZooKeeper 和 JournalNodes (一组 JournalNode), 而这两个组件完全部署时至少需要 3 或 3 个以上奇数节点, 故至少需要再投入 4 台机器。此处设计极为巧妙, 且需要大量独立机器进行备份, 这里只介绍方案, 不讲述细节。关于单点故障, 读者也可参考 MapR 公司的设计方案, 该公司对 HDFS 进行了重构, 不存在单点故障。

4. DataNode

HDFS 从进程, 集群中每台 slave 上都应当部署并长期运行此进程, DataNode 主要功能是管理所在机存储空间, DataNode 无法感知文件的存在, 其维护的都是块及其相关存储信息, 对外服务时, DataNode 会按照 Client 或 NameNode 的要求, 针对特定块进行读写和复制操作。

我们称部署 DataNode 的节点为存储从节点, 和 NameNode 不同, DataNode 应当有多个, 而且最好有 3 个以上。从物理存储角度看, DataNode 利用所在机某 (可设定) 特定存储空间来存放 HDFS 的块数据, 不过为确保数据安全, 本地用户访问此目录时, 数据已经经过加密。此外, 由于采用 master/slave 架构, 我们可以实时动态地向集群中增删 DataNode 节点。

5. 客户端

称欲使用集群存储服务的机器为 Client, 显然 Client 上至少应当有一套 NameNode 访问协议 (比如 NameNode 的 IP 和访问端口、文件存取协议等), 事实上, Client 上也只需要此类协议以及相关支持命令和 Jar 包, 并不需要任何驻守进程。当客户端使用 Shell 脚本访问集群时, Shell 会调用相关 Jar 包, Jar 里程序会根据约定规则主动访问 NameNode 并取得其服务。

下面, 简单讲解三个实体之间的关系, NameNode 管理所有 DataNode, 两者之间是管理和被管理关系, 不过要注意每次都是 DataNode 主动向 NameNode 发送心跳包, NameNode 并不会主动询问 DataNode。各个 DataNode 之间是对等的, 没有特别联系, 不过在块复制等操作时, DataNode 之间会有大量的 RPC 调用, 此时它们会在 NameNode 的指挥下协作完成数据存取任务。Client 端和 NameNode 之间是服务与被服务关系, NameNode 为 Client 提供服务, Client 则按使用者需求向 NameNode 发出相关存取请求。对于特定集群来说, NameNode 是固定的, Client 在不停地变化, 它可以是世界上任何一个节点。最后 Client 和 DataNode 之间更无关系可言, 实际上 Client 并不知道也不应当知道 DataNode 的存在, 只是在访问具体数据时, NameNode 会要求 Client 到某特定 DataNode 上进行存取。

从整体上看, NameNode 和 DataNode 之间是一个有机统一体, 它们屏蔽了关于集群

本身一切底层复杂架构，为 Client 和上层程序提供了统一的、透明的、海量的高性能存储服务。

4.2.2 HDFS 物理拓扑

根据 NameNode 安全机制的不同，HDFS 的物理拓扑主要分为典型拓扑和 HA 拓扑两大类。其中典型拓扑里只有一个 NameNode，可以选择 BackupNode 或 SecondaryNameNode 这两种不同机制来备份元数据。和单 NameNode 相比，HA 物理拓扑相对复杂，不仅需要添加至少 4 台机器，还涉及大量配置，不过这些都是值得的，因为当 Active NameNode 宕机后，HA 能够自动瞬间切换集群主节点，大大提升集群可靠性。

1. 典型拓扑

典型的 HDFS 物理拓扑包含一个 NameNode、一个 SecondaryNameNode 和若干个 DataNode，图 4-13 就是这样一个典型物理拓扑图，图中我们选定 cmaster0 部署 NameNode，cmaster1 部署 SecondaryNameNode，所有 cslavex 上部署 DataNode，iclientX 上部署 HDFS Client。

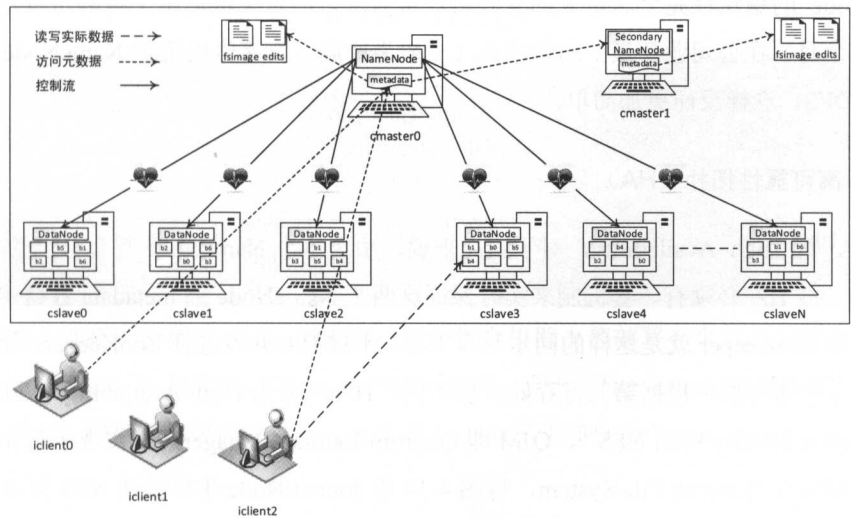


图 4-13 HDFS 典型物理拓扑图

为保持 NameNode 简单高效，尽量降低其设计的复杂性，NameNode 会以日志形式将用户（或程序）对 HDFS 的任何操作都追加到本地文件 fsimage 上。当此 NameNode 启动时（即启动或重启集群），它首先从存储在本地的镜像文件 fsimage 中读取 HDFS 状态信息，然后读取 edits 文件并将这些 edits 操作应用到内存中的 metadata 里。接着此

NameNode 将重启后的 HDFS 状态信息写入一个新的 fsimage 文件中, 并新建一个新的 edits 文件, 准备接受本次集群的 edits 信息, 也就是说, 系统中应当有多个 edits 和 fsimage 文件, 每启动一次 NameNode 就会生成一个新的 edits 和 fsimage, 只不过只有最新的 fsimage 和 edits 有效, 另外 NameNode 在集群启动时才会合并 fsimage 和 edits。

这个操作看似合理, 可正常情况下, NameNode 服务开启后就不应当再停止, 此时 NameNode 会一直将 edits 操作追加到同一个 edits 文件, fsimage 写入同一个 fsimage 文件, 这显然相当不合理, 因为一个稍微繁忙的集群就必然会导致单 edits 文件过大, 这不但增加了 HDFS 回滚、快照等机制的复杂性, 而且还会造成以后 NameNode 的启动越来越慢。

SecondaryNameNode 会定期合并 fsimage 和 edits 并且确保 edits 文件不会过大, 由于 SecondaryNameNode 和 NameNode 在 metadata 管理这块所需内存一样大, 通常将 SecondaryNameNode 部署在一台新机器上, 比如示例图中的 cmaster1。SecondaryNameNode 会在 cmaster1 上新建和 NameNode 一样的存储目录, 正常运行时, 它会定期合并存在 NameNode 的 fsimage 和 edits 文件, 并将合并结果存于此目录。NameNode 会根据需求, 随时取得此目录下的数据。

图 4-13 所示的 HDFS 物理拓扑也是当前工业界最常见的 HDFS 部署图, 诚然, 此时 NameNode 存在单点故障(一旦 NameNode 宕机, 整个 HDFS 将停止服务), 但当前 HDFS NameNode 的稳定性完全满足大部分公司日常需求, 而且集群运维工程师可以定期(如每隔 3 个月)在公司业务最少的时间点(一般为凌晨)直接停机维护 NameNode 甚至是整个 HDFS, 这样反而更加简单。

2. 高可靠性拓扑 (HA)

HA^[4]即 High Availability, 对于 HA 来说, 由于每个 NameNode 都有自己的 edits 和 fsimage, 故 HA 必须有一套机制来实时保证这两个 NameNode 的 metadata 数据一致。大数据组件 ZooKeeper 就是这样的同步互斥工具, 不过 HA 并没选择 ZooKeeper, 而是采用了第三方存储介质, 根据第三方存储介质不同, HA 可分为 High Availability With QJM^[5] 和 High Availability With NFS^[6], QJM 即 Quorum Journal Manager, 其实现示意如图 4-14 所示, NFS 即 Network File System, 将图 4-14 中 JournalNode 集群换成 NFS 即可。

就 QJM 和 NFS 比较而言, 官方推荐 QJM 机制, 这是因为 QJM 更加轻量级, 更加可控、高效。我们这里只讲解 HA with QJM, 正如图 4-14 所示, 在 cmaster0 上部署 NameNode, 在 cmaster3 上部署 NameNode, 其实这两个 NameNode 除了启动服务的 IP Address 不同外, 并无其他区别。接着在 cslavex 上部署 DataNode, 不过和以前不同的是, 所有 slave 都要连接到这两个 NameNode 上, 比如如下配置就是规定此 cslavex 同时连接到 cmaster0 和 cmaster3 上。

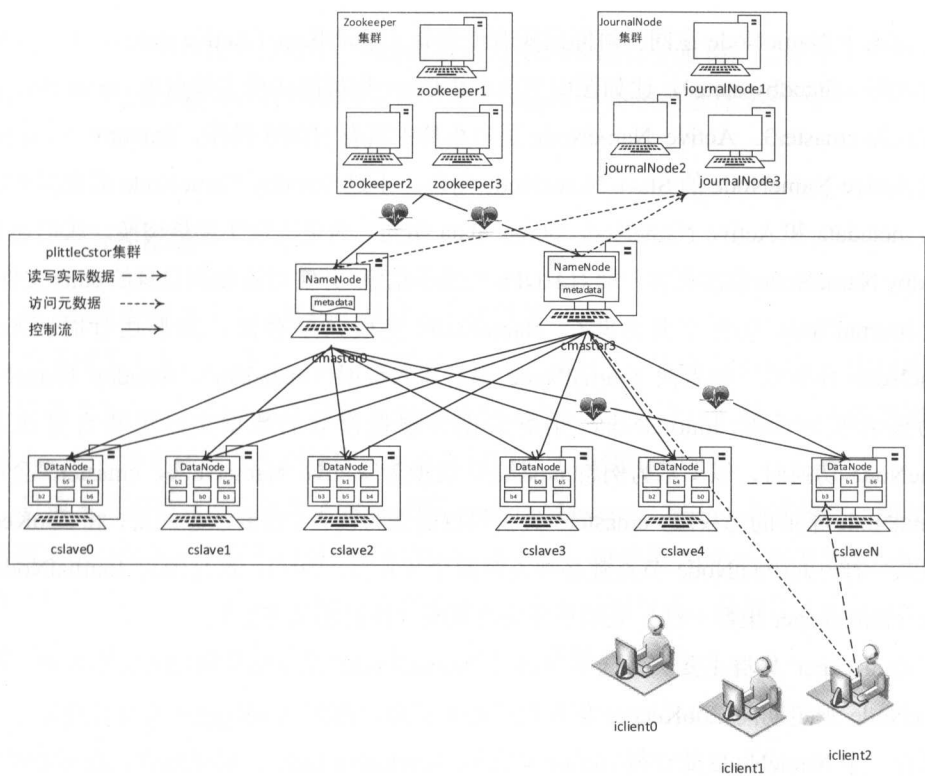


图 4-14 HA 示意图

```
<property>
  <name>dfs.namenode.rpc-address.njupt.nn1</name>
  <value>cmaster0.cloudlab.njupt.edu.cn:8020</value>
  .....
  <name>dfs.namenode.rpc-address.njupt.nn2</name>
  <value>cmaster3.cloudlab.njupt.edu.cn:8020</value>
</property>
```

接着，选择三台机器 JournalNode1~3，在每台机器上都部署一个 JournalNode，并配置这两个 NameNode 都指向这个 JournalNode 集群。到此，NameNode 双机热备的物理部署已经结束，接下来部署的 ZooKeeper 集群只是为了实现这两个 NameNode 之间自动热切换。

选择三台机器 ZooKeeper1~3，在这三台上都部署 ZooKeeper，配置 NameNode 指向这个 ZooKeeper 集群。至此 HA with QJM 部署结束，图中绘制的机器较多，在实际部署时可以将 3 个 ZooKeeper 服务和 3 个 JournalNode 服务放到相同的三台机器上，甚至这三台机器可以是集群中的某三个 slave，为清晰直观，编者绘制了 6 台不同的机器。

如图 4-14 所示，cslavex 上的 DataNode 主要负责管理本机块数据，此外每个 DataNode 都需要定期向 cmaster0、3 上的两个 NameNode 发送心跳包。

这两个 NameNode 在同一时间只能有一个处于活动状态 (Active state), 另一个处于待命状态 (Standby state), 比如图中当前 littleCstor 集群的活动主节点为 cmaster0, 待命主节点为 cmaster3。Active NameNode 负责集群中所有 HDFS 操作, Standby NameNode 充当 Active NameNode 的 Slave NameNode 角色, 不过 Standby NameNode 需要时刻保持它的 metadata 和 Active NameNode 的 metadata 相同, 为更快地实现热切换, 我们还要求 Standby NameNode 也接收客户端对 HDFS 的操作信息, 并实时追加到自身的 edits 文件里。

JournalNode 集群主要为两个 NameNode 实现共享存储, 集群运行时, Active NameNode 作为唯一写者向 JournalNode 集群写入数据 (metadata), Standby NameNode 作为读者实时读取 JournalNode 集群数据并根据读取的数据实时更新自身状态; NameNode 切换时, 以图示为例将 cmaster0 切换为 Active NameNode, cmaster0 会接管 JournalNode 集群的写权限, cmaster3 失去写权限变为读者。需要注意的是, 和 ZooKeeper 节点数一样, JournalNode 节点数必须为奇数个 (3, 5, 7...), 这是因为 JournalNode 集群内 (ZooKeeper 集群一样) 采用举手表决策略 (少数服从多数)。

ZooKeeper 集群主要负责选举 Active NameNode。这里实现机制其实很简单, 两个 NameNode 须定期向 ZooKeeper 集群发送心跳信息, 都到 ZooKeeper 集群处登记一下, 但只有一个 NameNode 能获得 Active 排它锁 (exclusive lock)。根据编者长期运维经验, 集群启动时, 先来者会获得该锁并成为 Active NameNode; 当 Active NameNode 崩溃或需要手工切换时, 以 cmaster0、3 为例, 假定 cmaster3 宕机, 此时 ZooKeeper 会瞬间检测到 cmaster3 已不在存活 (无心跳包), 此时 ZooKeeper 会将 exclusive lock 分配给 cmaster0。cmaster0 在获得 exclusive lock 和 JournalNode 集群写权限后, 接管整个集群。

需要指出的是, 双 NameNode 的 HA 只能实现一次自动热切, 即当 cmaster3 宕机后, ZooKeeper 会自动将 Active NameNode 瞬间切换至 cmaster0 上, 如果 cmaster0 再宕机, 且 cmaster3 未开启, 由于已经没有 Standby NameNode, 已无法自动热切。不过如果 cmaster3 已经修复, 则此时又可切换回 cmaster3。

3. 优劣比较

所有的事物都具有两面性, 拥有巨大优势的同时不可避免地存在一些缺陷, 单 NameNode 简单高效实用, 但存在单点故障。当 Active NameNode 宕机时, HA 能够自动切换 NameNode, 可是 HA 本身比较复杂, 加之分布环境复杂多变, 极有可能会出现诸如 “split-brain scenario”^[7] 这类很奇怪的问题。两者相比各有优缺, 无好坏之别, 只有适合与不适合之说, 单 NameNode 完全满足绝大部分场景, 不过若集群需要 24 小时 365 天不间断服务, 应当选用 HA。

从架构设计角度来看, HDFS 本身在设计上就存在缺陷, HBase 和 ZooKeeper 同样

是集群环境，却不存在单点故障，MapR 公司对 HDFS 进行了重构，已不存在单点故障，并且还保持了 API 兼容，有兴趣的读者可参考 MapR 公司相关设计文档^[8]。

4.2.3 HDFS 部署

HDFS 主要包含手工部署和工具部署两种部署方式。

当采用手工方式部署时，会加深部署者印象，不过手工部署的 HDFS 在用户权限分配、环境变量设置、服务启动设置、命令调用方式等方面存在巨大缺点，不但增加了维护成本，还可能直接导致上层程序因无充足权限而无法使用 HDFS。

目前主流的第三方工具为 Ambari 和 Cloudera Manager，和手工方式相比，部署工具则屏蔽大部分细节，不利于对组件架构的进一步理解，但使用 Ambari 部署的 HDFS 配置标准（权限、环境、启动方式等），兼容性强，不存在任何权限、环境等方面问题。由于 Ambari 为 Apache 社区开发并推荐，本书中我们只讲解 Ambari。

1. 手工部署 HDFS

作为 Hadoop 一部分，HDFS 相关 Jar 包和 Shell 命令集都合并到了 Hadoop 里，我们可以手动将 HDFS 相关资源从 Hadoop 中分离出来，也可以直接使用 Hadoop 包，不过在进行配置和启动时，只配置 HDFS，Hadoop 中 HDFS 的具体部署步骤如下：

Step1 制定部署规划

Step2 准备硬件机器和 OS 环境

Step3 配置单机 OS 环境和集群环境

Step4 解压 Hadoop、配置 HDFS

Step5 初始化 HDFS、测试 HDFS

2. 使用 Ambari 部署 HDFS

使用 Ambari 部署 HDFS 可以说是一键操作，难点几乎都在 Ambari 工具本身部署上，以下步骤从无到有，简单介绍了 Ambari 自身部署和使用 Ambari 部署 HDFS 的大概步骤：

Step1 制定部署规划

Step2 准备硬件机器和 OS 环境

Step3 配置单机 OS 环境和集群环境

Step4 部署 ambari-server

Step5 使用 ambari-server 部署 HDFS

例 2 请使用 Ambari 为 prelittleCstor 部署 HDFS。

解 由于大数据平台涉及太多组件，故部署之前最好制定一个完备的部署计划，否则极有可能导致各机角色分配过乱，甚至是部署失败。

本题以第 3 章为前提，只给出 HDFS 部署规划表（表 4-2）和部署效果图（图 4-15），具体部署过程请参见第 3 章。读者须注意，图 4-15 中 iclient0 到 cmaster0 或 cslave0、2 的链接（图中用 Ø 表示）实际上并不存在，也就是 iclient0 并不需要向任何机器汇报心跳包，只有当 iclient0 需要使用 HDFS 服务时，它才会主动链接 cmaster0 或 cslavex。

表 4-2 littleCstor 上 HDFS 部署规划

机 器	角 色	服 务
cmaster0	主服务	NameNode
cmaster1	主备份	SecondaryNameNode
cslave0	从服务	DataNode
cslave1		DataNode
cslave2		DataNode
cslave3		DataNode
cslave4		DataNode
cslave5		DataNode
iclient0	客户端	Client

使用 Ambari 部署 HDFS，Ambari 会自动新建 hdfs 用户和 hadoop 组，hdfs 用户是 HDFS 服务的默认超级用户，hdfs 用户则隶属 hadoop 组。

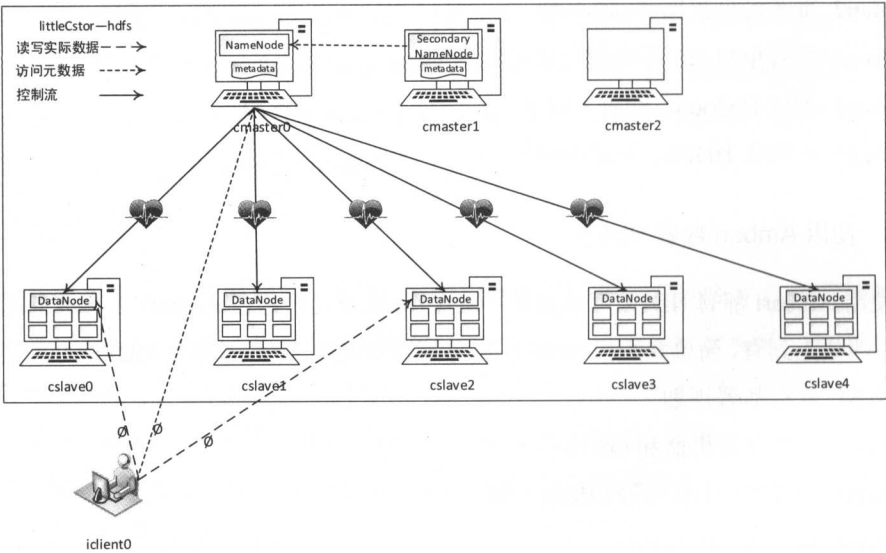


图 4-15 littleCstor—HDFS 效果图

4.2.4 HDFS 其他概念

HDFS 的主要设计目标之一就是在故障情况下也能保证数据存储的可靠性。HDFS 具备了较为完善的冗余备份和故障恢复机制，可以实现在集群中可靠地存储海量文件^[9]。

1. 硬件错误是常态而不是异常

HDFS 被设计为运行在普通硬件上，所以硬件故障是很正常的。HDFS 可能由成百上千的服务器节点构成，每个服务器节点上都存储着文件系统的部分数据，而 HDFS 的每个组件随时都有可能出现故障。因此，检测错误并快速自动恢复是 HDFS 的核心设计目标。

2. 流式数据访问

运行在 HDFS 上的应用主要以流式数据读取为主，做批量处理而不是用户交互处理，因此 HDFS 更关注数据访问的高吞吐量。

3. 大规模数据集

HDFS 的典型文件大小可能都在 GB 级甚至 TB 级，因此 HDFS 支持大文件存储，并能提供整体上高的数据传输带宽，能在一个集群里扩展到数百个节点。一个单一的 HDFS 实例应该能支撑数以千万计的文件。

4. 简单一致性模型

HDFS 的应用程序需要对文件实行一次性写、多次读的访问模式。文件一经创建、写入和关闭之后就不需要再更改了。这样的假定简化了数据一致性问题，使高吞吐量的数据访问成为可能。

5. 移动计算比移动数据更划算

对于大文件来说，移动计算比移动数据的代价要低。如果在数据旁边执行操作，那么效率会比较高，当数据特别大的时候效果更加明显，这样可以减少网络的拥塞和提高系统的吞吐量。这就意味着，把计算迁移到数据附近更好，而不是把数据传输到程序运行的地方。HDFS 提供了接口，以便让程序将自己移动到数据存储的地方执行。

6. 冗余备份

HDFS 将每个文件存储成一系列数据块 (Block)，默认块大小为 64MB (可配置)。为了容错，文件的所有数据块都会有副本 (副本数量即复制因子，可配置)。HDFS 的文件都是一次性写入的，并且严格限制为任何时候都只有一个写用户。DataNode 使用本地文件系统来存储 HDFS 的数据，但是它对 HDFS 的文件一无所知，只是用一个个文件存储 HDFS 的每个数据块。当 DataNode 启动的时候，它会遍历本地文件系统，产生一份 HDFS 数据块和本地文件对应关系的列表，并把这个报告发给 NameNode，这就是块报告 (Blockreport)。块报告包括了 DataNode 上所有块的列表。

7. 副本存放

HDFS 集群一般运行在多个机架上，不同机架上机器的通信需要通过交换机。通常情况下，副本的存放策略很关键，机架内节点之间的带宽比跨机架节点之间的带宽要大，它能影响 HDFS 的可靠性和性能。HDFS 采用机架感知 (Rack-aware) 的策略来改进数据的可靠性、可用性和网络带宽的利用率。通过机架感知，NameNode 可以确定每个 DataNode 所属的机架 ID。一般情况下，当复制因子是 3 的时候，HDFS 的部署策略是将一个副本放在同一机架上的另一个节点，一个副本存放在本地机架上的节点，最后一个副本放在不同机架上的节点。机架的错误远比节点的错误少，这个策略可以防止整个机架失效时数据丢失，不会影响到数据的可靠性和可用性，又能保证性能。图 4-16 体现了复制因子为 3 的情况下，各数据块的分布情况。

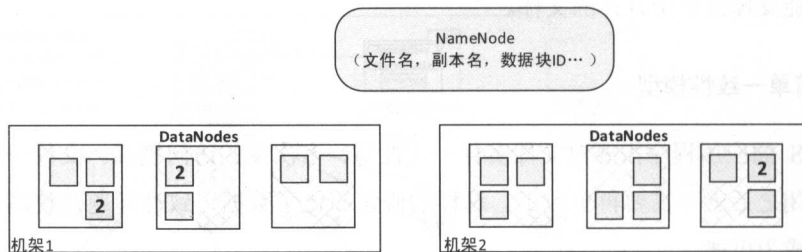


图 4-16 复制因子为 3 时数据块分布情况

8. 心跳检测

NameNode 周期性地从集群中的每个 DataNode 接受心跳包和块报告，NameNode 可以根据这个报告验证块映射和其他文件系统元数据。收到心跳包说明该 DataNode 工作正常。如果 DataNode 不能发送心跳消息，NameNode 会标记最近没有心跳的 DataNode 为宕机，不会发给它们任何新的 I/O 请求。这样一来，任何存储在宕机的 DataNode 的数据

将不再有效。DataNode 的宕机会造成一些数据块的副本数下降并低于指定值，NameNode 会不断检测这些需要复制的数据块，并在需要的时候重新复制。引发重新复制有多种原因：DataNode 不可用、数据副本的损坏、DataNode 上的磁盘错误或者复制因子增大。

9. 安全模式

系统启动时，NameNode 会进入一个安全模式。此时不会出现数据块的写操作。NameNode 会收到各个 DataNode 拥有的数据块列表对的数据块报告，由此 NameNode 获得所有的数据块信息。数据块达到最小副本数时，该数据块就被认为是安全的。在一定比例（可配置）的数据块被 NameNode 检测确认是安全之后，再等待若干时间，NameNode 自动退出安全模式状态。当检测到副本数不足的数据块，该块会被复制到其他数据节点，以达到最小副本数。

10. 数据完整性检测

多种原因会造成从 DataNode 获取的数据块有可能是损坏的。HDFS 客户端软件实现了对 HDFS 文件内容的校验和（Checksum）检查，在 HDFS 文件创建时，会计算每个数据块的校验和，并将校验和作为一个单独的隐藏文件保存在命名空间下。当客户端获取文件后，它会检查从 DataNode 获得的数据块对应的校验和是否和隐藏文件中的相同，如果不同，客户端就会认为数据块有损坏，将从其他 DataNode 获取该数据块的副本。

11. 空间回收

文件被用户或应用程序删除时，并不是立即就从 HDFS 中移走，而是先把它移动到 /trash 目录里。只要还在这个目录里，文件就可以被迅速恢复。文件在这个目录里的时间是可以配置的，超过了这个时间，系统就会把它从命名空间中删除。文件的删除操作会引起相应数据块的释放，但是从用户执行删除操作到从系统中看到剩余空间的增加可能会有一个时间延迟。只要文件还在 /trash 目录里，用户可以取消删除操作。当用户想取消时，可以浏览这个目录并取回文件，这个目录只保存被删除文件的最后副本。这个目录还有一个特性，就是 HDFS 会使用特殊策略自动删除文件。当前默认的策略是：文件超过 6 个小时后自动删除，在未来版本里，这个策略可以通过定义良好的接口来配置。

12. 元数据磁盘失效

映像文件和事务日志是 HDFS 的核心数据结构。如果这些文件损坏，将会导致 HDFS 不可用。NameNode 可以配置为支持维护映像文件和事务日志的多个副本，任何对映像文件或事务日志的修改，都将同步到它们的副本上。这样会降低 NameNode 处理命名空

间事务的速度,然而这个代价是可以接受的,因为 HDFS 是数据密集,而非元数据密集的。当 NameNode 重新启动的时候,总是选择最新的一致映象文件和事务日志。在 HDFS 集群中 NameNode 是单点存在的,如果它出现故障,必须手动干预。

13. 快照

快照支持存储某个时间的数据复制,当 HDFS 数据损坏的时候,可以回滚到过去一个已知正确的时间点。当前的 HDFS 支持完备的快照功能。

4.3 HDFS 接口

1. 接口综述

作为大数据处理领域最基础的存储服务,HDFS 针对不同的上层应用,提供了四类统一访问接口,分别为:

- HDFS 自带 Web 接口
- HDFS Shell 接口
- WebHDFS REST API 接口
- HDFS Java API 接口

实际上,HDFS 还提供了诸如“C API libhdfs”等特殊功能接口,不过本书重点讲解 Shell 和 Java 接口,至于类似 C、本地库等访问方式,请读者参考文献[10]。

HDFS 自带的 Web 页面主要面向 HDFS 管理员,在此页面上显示的信息主要包括 HDFS 系统级别统计信息和文件系统,且此页面只支持读,不支持写操作。

WebHDFS REST API 接口面向前端工程师(页面开发工程师),其遵循 CRUD 原则(Create、Read、Update、Delete),支持 HTTP GET、PUT、POST 和 DELETE 四大协议。通过调用此 API,前端工程师能够开发一套自定义 Web 页面。和自带管理员页面只读功能不同,在页面上可以向 HDFS 写数据。

Shell 接口主要针对 HDFS 管理员,通过 Shell 接口,管理员能够查看 HDFS 系统级别统计信息和文件系统。熟悉数据驱动型编程的读者应当知道,不同的程序之间通常都通过 Shell 或 Python 脚本连接,故 Shell 接口也面向后台开发工程师、数据工程师、算法工程师。

从名字就可以看出,HDFS Java API 面向 Java 开发工程师,细分为后台开发工程师、数据工程师、算法工程师。

2. 实战 HDFS Web

由于 Web 接口内容较少，下面直接讲述该内容，在 brilliant 机上的 FireFox 地址栏中输入“cmaster0.cloudlab.njupt.edu.cn:50070”，打开该地址可得图 4-17 所示界面，图 4-18～图 4-21 展示了 HDFS 中 mynjupt.txt 文件共占用两个块，每个块都有两个副本，且各副本分别存于不同机器。



图 4-17 HDFS Web 接口主页

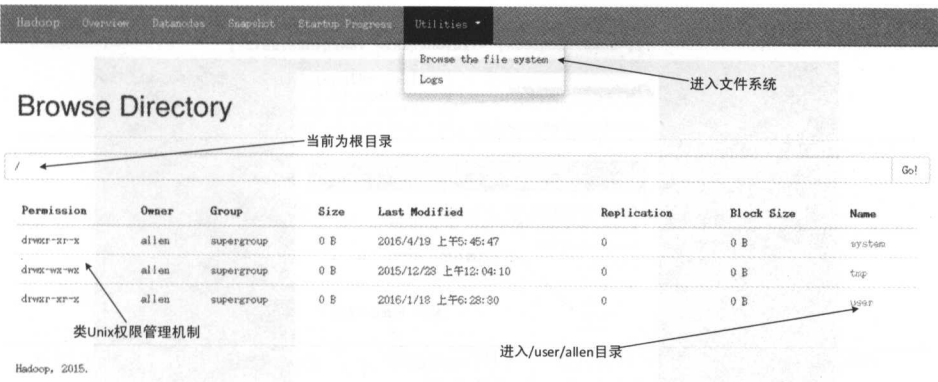


图 4-18 HDFS Web 文件系统接口

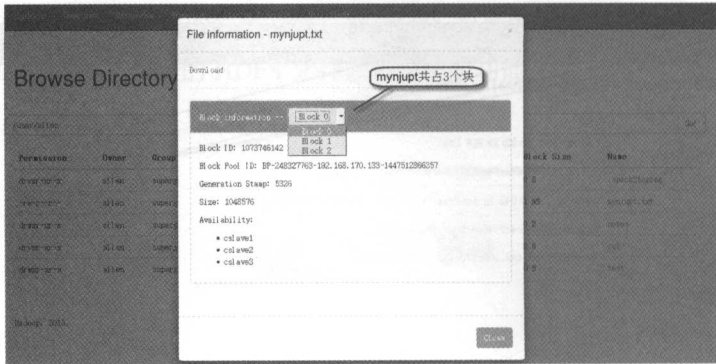
Browse Directory

文件mynjupt.txt大小为2.16M

块大小为1M → 故该文件应当占用3个块 (图19b)

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
drwxr-xr-x	allen	supergroup	0 B	2015/11/17 下午4:58:27	0	0 B	.sparkStaging
-rwxr-xr-x	allen	supergroup	2.16 MB	2016/4/20 下午8:00:14	3	1 MB	mynjupt.txt
drwxr-xr-x	allen	supergroup	0 B	2015/11/17 下午4:51:48	0	0 B	spites

(a)



(b)

图 4-19 mynjupt 块和文件大小

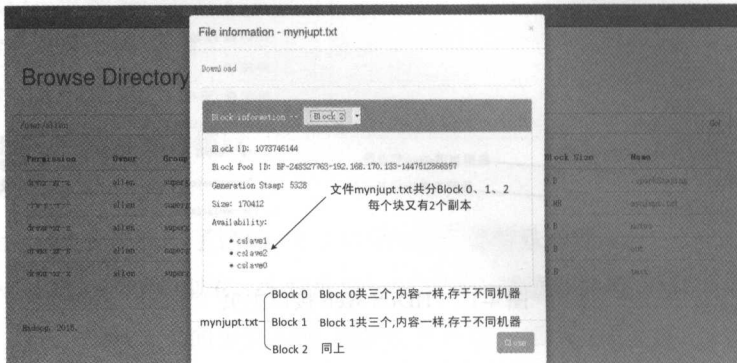


图 4-20 mynjupt 块 Block2 位置信息

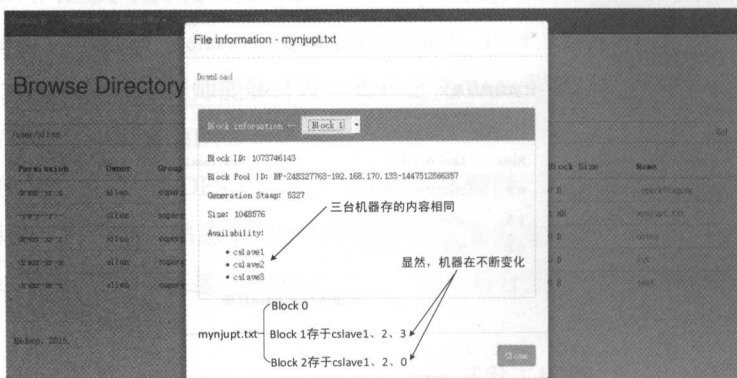


图 4-21 mynjupt 块 Block1 位置信息

至此 HDFS 自带 Web 界面主要功能展示完毕，待读者搭建好 littleCstor 后，请读者仔细研读此界面上展示的所有信息。

4.4 实战 HDFS Shell

Shell 接口是管理 HDFS 的最简单接口，也应当是读者最先接触的接口，通过 Shell 命令行，能够很方便查看 HDFS 文件系统、HDFS 系统级别统计信息和数据管理工具。本节讲述通过 Shell 来操作 HDFS。

4.4.1 HDFS 文件级命令集

HDFS 本质上是一个用来存储文件的存储器，不过由于其分布式特性，在 HDFS 中存储文件和在普通环境下又有所不同。下面通过实例操作 HDFS 文件系统，这里的文件系统指的是通过 Shell 命令行操作 HDFS 里的文件或文件夹，比如查看某文件、新建文件夹、删除文件文件等操作。

在 brilliant 上，借助软件 SecureCRT 进入 iclient0 客户端，在 iclient0 上，以 allen 用户执行命令“hdfs dfs”，命令行罗列了（图 4-22）当前版本 HDFS 支持的所有 dfs 操作：

```
[allen@iclient0 ~]$ hdfs dfs
Usage: hadoop fs [generic options]
[-appendToFile <localsrc> ... <dst>]
[-cat [-ignoreCrc] <src> ...]
[-checksum <src> ...]
[-chgrp [-R] GROUP PATH...]
[-chmod [-R] <MODE[,MODE]...> [OCTALMODE] PATH...]
[-chown [-R] [OWNER][:[GROUP]] PATH...]
[-copyFromLocal [-f] [-p] [-l] <localsrc> ... <dst>]
[-copyToLocal [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
[-count [-q] [-h] <path> ...]
[-cp [-f] [-p] [-p[topax]] <src> ... <dst>]
[-createSnapshot <snapshotDir> [<snapshotName>]]
[-deleteSnapshot <snapshotDir> <snapshotName>]
[-df [-h] [<path> ...]]
[-du [-s] [-h] <path> ...]
The general command line syntax is
bin/hadoop command [genericOptions] [commandOptions]

[allen@iclient0 ~]$
```

图 4-22 当前 HDFS 支持的 dfs 操作

1. 新建文件

下面的操作完成在 iclient0 上新增 allen 用户；在 HDFS 上新建“/user/allen”目录并设置此目录所有者为用户 allen；设置目录“/user/allen”为 allen 默认目录（图 4-23）。

在 brilliant 上，借助 SecureCRT 远程登录 iclient0，下述操作都在 iclient0 上。

Step1 进入 root 用户, 执行“adduser allen”命令添加用户 allen, 执行完后, 使用“passwd allen”设置 allen 用户密码。

Step2 使用命令“su - hdfs”, 完成从 root 切换至用户 hdfs, 此用户为 HDFS 的超级用户。执行命令“hdfs dfs -mkdir /user/allen”, 此命令完成在 HDFS 里新建目录“/user/allen”。执行完后, 目录“/user/allen”默认拥有者为其创建者 hdfs, 默认所在组也为 hdfs。

Step3 依旧使用超级用户 hdfs, 执行命令“hdfs dfs -chown -R allen /user/allen”, 将目录“/user/allen”所有者改为 allen。

Step4 上述命令执行后, “/user/allen”目录会自动成为用户 allen 家目录。上述命令执行过程如图 4-23 所示。

经过上述四步后, 可将 hdfs 中新建的“/user/allen”更改为 allen 用户主目录。

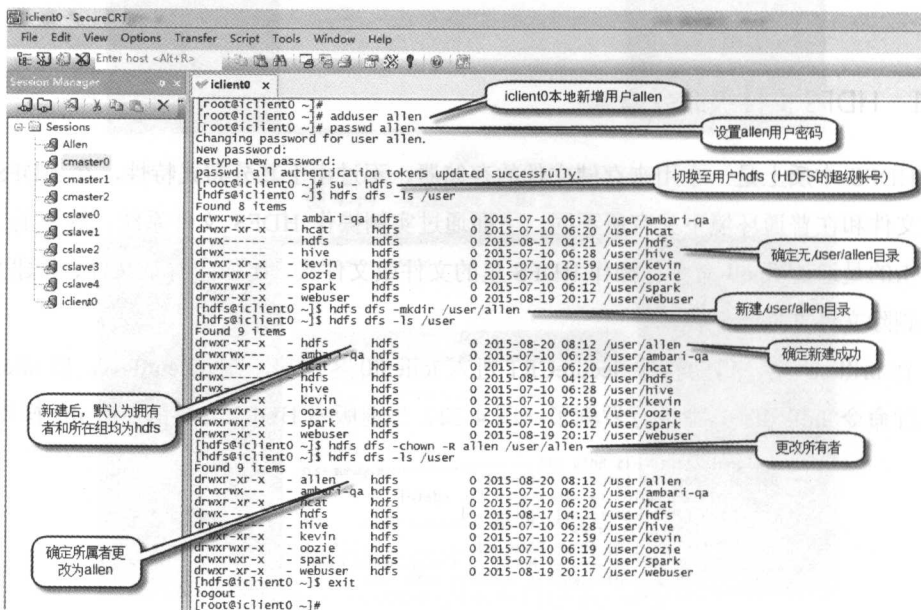


图 4-23 HDFS 新建用户 allen

2. 上传文件

下面的操作完成以 allen 身份, 将 iclient0 上本地文件 happy.txt 上传至 HDFS 的“/user/allen/”目录下; 使用 dfs -cat 命令查看上传后的文件内容。

①切换至 allen 用户, iclient0 上新建本地文件 happy.txt 并写入如下内容:

```

I am very happy.
njupt is very happy.
  
```

保存后, 执行“hdfs dfs -put happy.txt /user/allen”, 此命令完成将本地文件 happy.txt 上传至 HDFS 的“/user/allen”目录下。

② 执行“`hdfs dfs -cat /user/allen/happy.txt`”，此目录用来查看 HDFS 里文件“`/user/allen/happy.txt`”，注意整个过程都是 allen 用户，图 4-24 为执行过程。

```
[allen@iclient0 ~]$ cat happy.txt
I am very happy.
njupt is very happy.
[allen@iclient0 ~]$ hdfs dfs -put happy.txt /user/allen
[allen@iclient0 ~]$ hdfs dfs -cat /user/allen/happy.txt
I am very happy.
njupt is very happy.
[allen@iclient0 ~]$
```

查看iclient0本地文件
将该本地文件上传至HDFS
查看HDFS中刚上传的文件

图 4-24 上传文件

3. 其他常见文件操作

下面以 allen 用户，使用 HDFS 里文件“`/user/allen/happy.txt`”，练习追加、复制、重命名、查看文件大小、查看文件权限等各种文件操作（见图 4-25）。

```
[allen@iclient0 ~]$ cat someWord
aaa bbb
ccc ddd
[allen@iclient0 ~]$ hdfs dfs -appendToFile someWord happy.txt
[allen@iclient0 ~]$ hdfs dfs -cat /user/allen/happy.txt
I am very happy.
njupt is very happy.
aaa bbb
ccc ddd
[allen@iclient0 ~]$ hdfs dfs -cp happy.txt ihappy.txt
[allen@iclient0 ~]$ hdfs dfs -cat ihappy.txt
I am very happy.
njupt is very happy.
aaa bbb
ccc ddd
[allen@iclient0 ~]$ hdfs dfs -mv ihappy.txt vhappy.txt
[allen@iclient0 ~]$ hdfs dfs -du -s -h vhappy.txt
54  vhappy.txt
[allen@iclient0 ~]$
```

iclient0本地文件someWord
将本地文件someWord追加到HDFS的文件happy.txt里
查看追加效果
复制命令
查看是否复制成功
重命名
查看HDFS里文件大小

图 4-25 其他操作

4.4.2 HDFS 系统级命令集

HDFS 系统级信息指的是针对整个 HDFS 系统的统计信息，比如当前 DataNode 个数、集群总存储空间、剩余空间、块功能检测等，图 4-26 为当前版本 HDFS 支持的所有 `dfsadmin` 操作。


```

[allen@iclient0 ~]$ hdfs dfsadmin
Usage: hdfs dfsadmin
Note: Administrative commands can only be run as the HDFS superuser.
    [-report [-live] [-dead] [-decommissioning]]
    [-safemode <enter | leave | get | wait>]
    [-saveNamespace]
    [-rollEdits]
    [-restoreFailedStorage true|false|check]
    [-refreshNodes]
    [-setQuota <quota> <dirname>...<dirname>]
    [-clrQuota <dirname>...<dirname>]
    [-setSpaceQuota <quota> [-storageType <storageType>] <dirname>...<dirname>]
    [-clrSpaceQuota [-storageType <storageType>] <dirname>...<dirname>]
    [-finalizeUpgrade]
    [-rollingUpgrade [<query>|prepare|finalize>]]
    [-refreshServiceAcl]
    [-refreshUserToGroupsMappings]
    [-refreshSuperUserGroupsConfiguration]
    [-refreshCallQueue]
    [-refresh <host:ipc_port> <key> [arg1..argn]]
    [-reconfig <datanode>[...> <host:ipc_port> <start|status>]
    [-printTopology]
    [-refreshNameNodes datanode_host:ipc_port]
    [-deleteBlockPool datanode_host:ipc_port blockpoolId [force]]
    [-setBalancerBandwidth <bandwidth in bytes per second>]
    [-fetchImage <local directory>]
    [-allowSnapshot <snapshotDir>]
    [-disallowSnapshot <snapshotDir>]
    [-shutdownDatanode <datanode_host:ipc_port> [upgrade]]
    [-getDatanodeInfo <datanode_host:ipc_port>]
    [-metasave filename]
    [-triggerBlockReport [-incremental] <datanode_host:ipc_port>]
    [-help [cmd]]

Generic options supported are
-conf <configuration file>      specify an application configuration file
-D <property=value>              use value for given property
-fs <local|namenode:port>        specify a namenode
-jt <local|resourcemanager:port> specify a ResourceManager
-files <comma separated list of files> specify comma separated files to be copied to the m
ap reduce cluster
-libjars <comma separated list of jars> specify comma separated jar files to include in th
e classpath.
-archives <comma separated list of archives> specify comma separated archives to be unarch
ived on the compute machines.

The general command line syntax is
bin/hadoop command [genericOptions] [commandOptions]

[allen@iclient0 ~]$

```

HDFS系统级管理命令集

这些命令主要用于管理HDFS服务，而不是管理HDFS存储空间

图 4-26 HDFS 管理员命令

dfsadmin 命令集主要用来管理 HDFS 系统级信息，比如可以用 report 命令查看集群当前节点运行状态，可以手工让集群进入安全模式等，下面演示几个重要常用命令。

从图 4-27 可以看出，report 选项用来显示集群状态信息，如集群总空间、剩余空间、集群当前活动节点数，以及每个节点统计信息等。

在刚开启 HDFS 时，NameNode 需要统计并合并各个 DataNode 汇报过来的块信息，为保证数据安全，NameNode 将自身设置成一种只读模式，称此只读模式为安全模式。读者不必担心，NameNode 会自动进出 safemode，之所以提供手工设置选项，是由于在某些极端情况下（DataNode 突然大面积死亡），管理员需要强制设置集群进入安全模式。图 4-28 演示了进出 safemode 命令，请读者务必练习。

```

[hdfs@client0 ~]$ hdfs dfsadmin -report
Configured Capacity: 258183639040 (240.45 GB)
Present Capacity: 219728968810 (204.64 GB)
DFS Remaining: 217848160256 (202.89 GB)
DFS Used: 1880808554 (1.75 GB)
DFS Used%: 0.86%
Under replicated blocks: 0
Blocks with corrupt replicas: 0
Missing blocks: 0

-----
Live datanodes (5):

Name: 10.10.201.107:50010 (cslave3.cloudLab.njupt.edu.cn)
Hostname: cslave3.cloudLab.njupt.edu.cn
Decommission Status : Normal
Configured Capacity: 51636727808 (48.09 GB)
DFS Used: 451742746 (459.43 MB)
Non DFS Used: 7284088934 (6.78 GB)
DFS Remaining: 43870896128 (40.86 GB)
DFS Used%: 0.93%
DFS Remaining%: 84.96%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 6
Last contact: Sat Aug 22 20:17:49 EDT 2015

Name: 10.10.201.104:50010 (cslave1.cloudLab.njupt.edu.cn)
Hostname: cslave1.cloudLab.njupt.edu.cn
Decommission Status : Normal
Configured Capacity: 51636727808 (48.09 GB)
DFS Used: 377585460 (360.09 MB)
Non DFS Used: 7810818252 (7.27 GB)
DFS Remaining: 43448324096 (40.46 GB)
DFS Used%: 0.73%
DFS Remaining%: 84.14%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 6
Last contact: Sat Aug 22 20:17:49 EDT 2015

Name: 10.10.201.103:50010 (cslave0.cloudLab.njupt.edu.cn)
Hostname: cslave0.cloudLab.njupt.edu.cn
Decommission Status : Normal
Configured Capacity: 51636727808 (48.09 GB)
DFS Used: 296336052 (284.52 MB)
Non DFS Used: 7813746892 (7.28 GB)
DFS Remaining: 43524644864 (40.54 GB)
DFS Used%: 0.58%
DFS Remaining%: 84.29%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 6
Last contact: Sat Aug 22 20:17:49 EDT 2015

Name: 10.10.201.105:50010 (cslave2.cloudLab.njupt.edu.cn)
Hostname: cslave2.cloudLab.njupt.edu.cn
Decommission Status : Normal
Configured Capacity: 51636727808 (48.09 GB)
DFS Used: 340598682 (324.82 MB)
Non DFS Used: 7652184166 (7.13 GB)
DFS Remaining: 43643944960 (40.65 GB)
DFS Used%: 0.66%
DFS Remaining%: 84.52%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 4
Last contact: Sat Aug 22 20:17:48 EDT 2015

Name: 10.10.201.109:50010 (cslave4.cloudLab.njupt.edu.cn)
Hostname: cslave4.cloudLab.njupt.edu.cn
Decommission Status : Normal
Configured Capacity: 51636727808 (48.09 GB)
DFS Used: 382545614 (364.82 MB)
Non DFS Used: 7893831986 (7.35 GB)
DFS Remaining: 43360350208 (40.38 GB)
DFS Used%: 0.74%
DFS Remaining%: 83.97%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 6
Last contact: Sat Aug 22 20:17:47 EDT 2015

[hdfs@client0 ~]$

```

图 4-27 littleCstor 集群状态信息

图 4-29 演示了 refreshNodes、printTopology 选项, 至于其他命令, 功能不尽相同, 比如当编者将 littleCstor 的 hdp-2.2.6 升级至 hdp-2.6 时, 就需要用到 rollingUpgrade、rollEdits; 选项 refreshServiceAcl 则用于刷新访问控制配置。显然, 有些命令不应当在集群上轻易执行 (deleteBlockPool), 编者还是建议读者练习所有命令, 哪怕是最不起眼的命令。

```
iclient0 x
[hdfs@iclient0 ~]$ hdfs dfsadmin -safemode get
Safe mode is OFF
[hdfs@iclient0 ~]$ hdfs dfsadmin -safemode enter
Safe mode is ON
[hdfs@iclient0 ~]$ hdfs dfsadmin -safemode get
Safe mode is ON
[hdfs@iclient0 ~]$ hdfs dfsadmin -safemode leave
Safe mode is OFF
[hdfs@iclient0 ~]$ hdfs dfsadmin -safemode get
Safe mode is OFF
[hdfs@iclient0 ~]$
```

图 4-28 HDFS 安全模式

```
[hdfs@iclient0 ~]$ hdfs dfsadmin -refreshNodes
Refresh nodes successful
[hdfs@iclient0 ~]$ hdfs dfsadmin -printTopology
Rack: /default-rack
10.10.201.103:50010 (cslave0.cloudlab.njupt.edu.cn)
10.10.201.104:50010 (cslave1.cloudlab.njupt.edu.cn)
10.10.201.105:50010 (cslave2.cloudlab.njupt.edu.cn)
10.10.201.107:50010 (cslave3.cloudlab.njupt.edu.cn)
10.10.201.109:50010 (cslave4.cloudlab.njupt.edu.cn)
[hdfs@iclient0 ~]$
```

图 4-29 其他 HDFS 管理命令举例

前面讲述的 dfs 和 dfsadmin 是 HDFS Shell 里最常用的功能, 掌握这两个命令集几乎能够胜任日常一切工作。但前文讲述的部分功能 (如 ha 管理、元数据管理) 上述两个命令集并未涉及, 下面编者即讲述这部分命令。

HDFS 支持的所有 Shell 操作入口皆为 hdfs 命令, 之所以优先讲述 dfs 命令集是因为操作 dfs 时, 用户可以直观看到 HDFS 里的文件 (如/user/allen/happy.txt), 方便理解。此处的操作则更为抽象。

1. 命令简介

从图 4-30 可以看出, 命令 hdfs 是所有 HDFS Shell 的入口命令, 不过除了 dfs 和 dfsadmin 较为“真实”、常用外, 其他命令则较为“抽象”、少用。比如只有当集群开启 ha 服务时, haadmin、zkfc、nfs3、journalnode 才可使用。又比如 oiv 用户离线处理 metadata 数据 (NameNode 元数据)。由于命令太多, 请读者自行参与帮助文档。

(1) namenode-format

dfs 和 dfsadmin 已经讲述。选项 namenode-format 切记不可执行, 否则整个集群数据荡然无存。手工搭建集群时, 集群初次启动之前须执行 namenode-format 格式化主节点, 不过使用 Ambari 部署 HDP 时, 内部脚本自动执行了本步骤。极端情况, 当集群发生灾难性破坏的情况下, 管理员可执行此命令格式化整个集群 (此命令会自动删除所有元数据)。



图 4-30 hdfs 命令集

(2) secondarynamenode

选项 secondarynamenode 为启动 metadata 备份服务，读者可练习此命令，不过在 littleCstor 集群中尽量不要轻易执行（因为按规划 secondarynamenode 只在 cmaster1 上），此外 secondarynamenode 服务命令可能会引起读者误解，其并不是第二个 namenode，就算是 namenode 损坏，secondarynamenode 也永远做不成 namenode，其实质是合并 namenode 的 metadata 独立进程。

(3) namenode

选项 namenode 为启动 namenode 服务命令，读者可练习此命令，不过在 littleCstor 上还是尽量不要执行（因为按规划 namenode 只在 cmaster0 上）。

(4) datanode

选项 datanode 同 namenode，为启动本机 datanode 服务，littleCstor 运行正常时，不要执行。

(5) 选项 haadmin、zkfc、nfs3、journalnode 前面已经讲述，当集群开启 ha 时才可使用。

(6) fsck

选项 fsck 属于常用命令，其主要用来检查 HDFS 块健康状态信息，读者可从图 4-32 中看到其执行效果图。

(7) balancer

选项 balancer 属于常用命令, 用来平衡整个集群内各个 DataNode 上的块分布, 比如当 cs1ave0 上块数量太多而 cs1ave4 上又太少时, balancer 会自动将多余的块分配到 cs1ave4 上, 当集群添加节点时, 请务必手工执行此命令 (图 4-31)。

(8) 选项 oiv 和 oev

它们非常重要, 当集群发生灾难性破坏时, 需要用到这两个命令恢复 metadata。

(9) 其他

如 fetchdt、group 命令等也经常使用, 请读者自行分析。

```
[allen@iclient0 ~]$ hdfs version 查看HDFS版本
Hadoop 2.7.1
Subversion https://git.wip-us.apache.org/repos/asf/hadoop.git -r 15ecc87ccf4a0228f35af08fc56de536e6ce657a
Compiled by jenkins on 2015-06-29T06:04Z
Compiled with protoc 2.5.0
From source with checksum fc0a1a23fc1868e4d5ee7fa2b28a58a
This command was run using /home/allen/hadoop-2.7.1/share/hadoop/common/hadoop-common-2.7.1.jar
[allen@iclient0 ~]$
[allen@iclient0 ~]$ hdfs balancer ** balancer命令
16/04/18 14:45:46 INFO balancer.Balancer: namenodes = [hdfs://cmaster0:8020]
16/04/18 14:45:46 INFO balancer.Balancer: parameters = Balancer.Parameters[BalancingPolicy.Node, threshold=10.0,
max idle iteration = 5, number of nodes to be excluded = 0, number of nodes to be included = 0]
Time Stamp      Iteration#  Bytes Already Moved  Bytes Left To Move  Bytes Being Moved
16/04/18 14:45:49 INFO net.NetworkTopology: Adding a new node: /default-rack/192.168.170.137:50010
16/04/18 14:45:49 INFO net.NetworkTopology: Adding a new node: /default-rack/192.168.170.136:50010
16/04/18 14:45:49 INFO net.NetworkTopology: Adding a new node: /default-rack/192.168.170.135:50010
16/04/18 14:45:49 INFO net.NetworkTopology: Adding a new node: /default-rack/192.168.170.134:50010
16/04/18 14:45:49 INFO balancer.Balancer: 0 over-utilized: []
16/04/18 14:45:49 INFO balancer.Balancer: 0 underutilized: []
The cluster is balanced. Exiting...
Apr 18, 2016 2:45:49 PM      0      0 B      0 B      -1 B
Apr 18, 2016 2:45:49 PM Balancing took 2.748 seconds
[allen@iclient0 ~]$ hdfs oev -p xml -i ~/edits -o ~/njuptEdits ** edits里保存了
[allen@iclient0 ~]$ cat njuptEdits 集群文件所有变更信息
文件太大, 输出太多
```

图 4-31 version 和 balancer 命令

由于 littleCstor 的 namenode 部署在 cmaster0 上, 故图中 oev 可以执行的前提是已经将 cmaster0 上的文件 edits_00000000000000000001-0000000000000005386 拷贝至 iclient0 上。编者的 littleCstor 集群存储数据较少, balancer 命令检测到块分布均匀, 无须平衡。

2. fsck

选项 fsck 为常用命令, 默认情况下, HDFS 中的每个块数据都有两个备份, 也就是相同的数据块集群中会存储三个, 当某个块损坏时, 须使用此命令检测并移除, 以确保整个集群健康运行 (图 4-32 和图 4-33)。

图 4-34 为在 littleCstor 上, 使用目录/user/allen 下的文件, 执行命令 list-corruptfileblocks、blocks 和 racks (使用整个文件系统"/") 的输出过程。

HDFS Shell 接口是使用 HDFS 服务最简单、最常用的接口, 也是读者最先接触的接口, 通过 hdfs dfs 提供的命令集, 用户可以管理存储在 HDFS 中的具体文件 (包括目录文件); 通过 hdfs dfsadmin, 管理员可管理 HDFS 各个节点信息, 集群信息。请读者务必掌握这两大命令集, 至于 hdfs 下面的命令集 (dfs 和 dfsadmin 除外), 请尽量不要在真实集群上操作, 如需学习, 可自建小集群练习这几个命令。

```
[allen@iclient0 ~]$ hdfs fsck
Usage: hdfs fsck <path> [-list-corruptfileblocks | [-move | -delete | -openforwrite] [-files [-blocks [-locations
| -racks]]]]
    <path> start checking from this path
    -move move corrupted files to /lost+found
    -delete delete corrupted files
    -files print out files being checked
    -openforwrite print out files opened for write
    -includeSnapshots include snapshot data if the given path indicates a snapshottable directory or th
ere are snapshottable directories under it
    -list-corruptfileblocks print out list of missing blocks and files they belong to
    -blocks print out block report
    -locations print out locations for every block
    -racks print out network topology for data-node locations
    -storagepolicies print out storage policy summary for the blocks

    -blockid print out which file this blockId belongs to, locations (nodes, racks) of this block, and
other diagnostics info (under replicated, corrupted or not, etc)

Please Note:
    1. By default fsck ignores files opened for write, use -openforwrite to report such files. They are usual
ly tagged CORRUPT or HEALTHY depending on their block allocation status
    2. Option -includeSnapshots should not be used for comparing stats, should be used only for HEALTH check,
as this may contain duplicates if the same file present in both original fs tree and inside snapshots.
[allen@iclient0 ~]$
```

磁盘检查与维护
命令统一入口

图 4-32 fsck 所有命令集

```
[allen@iclient0 ~]$ hdfs fsck /user/allen/happySign.mp3 -locations
Connecting to namenode via http://cmaster0:50070/fsck?ugi=all&locations=1&path=%2Fuser%2Fallen%2FhappySign.mp3
FSCK started by allen (auth:SIMPLE) from /192.168.170.138 for path /user/allen/happySign.mp3 at Mon Apr 18 15:14:
31 PDT 2016
.....Status: HEALTHY
Total size: 3289431 B
Total dirs: 0
Total files: 1
Total symlinks: 0
Total blocks (validated): 1 (avg. block size 3289431 B)
Minimally replicated blocks: 1 (100.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 0 (0.0 %)
Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 3
Average block replication: 3.0
Corrupt blocks: 0
Missing replicas: 0 (0.0 %)
Number of data-nodes: 4
Number of racks: 1
FSCK ended at Mon Apr 18 15:14:31 PDT 2016 in 12 milliseconds

The filesystem under path '/user/allen/happySign.mp3' is HEALTHY
[allen@iclient0 ~]$
```

检查happySign.mp3
文件是否受侵蚀

检测结果: 健康

图 4-33 locations 选项及其输出

```
[allen@iclient0 ~]$ hdfs fsck /user/allen -list-corruptfileblocks
Connecting to namenode via http://cmaster0:50070/fsck?ugi=all&listcorruptfileblocks=1&path=%2Fuser%2Fallen
The filesystem under path '/user/allen' has 0 CORRUPT files
[allen@iclient0 ~]$ hdfs fsck /user/allen -blocks
Connecting to namenode via http://cmaster0:50070/fsck?ugi=all&blocks=1&path=%2Fuser%2Fallen
FSCK started by allen (auth:SIMPLE) from /192.168.170.138 for path /user/allen at Mon Apr 18 15:20:49 PDT 2016
.....Status: HEALTHY
Total size: 3314449 B
Total dirs: 7
Total files: 20
Total symlinks: 0
Total blocks (validated): 19 (avg. block size 174444 B)
Minimally replicated blocks: 19 (100.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 0 (0.0 %)
Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 3
Average block replication: 3.0
Corrupt blocks: 0
Missing replicas: 0 (0.0 %)
Number of data-nodes: 4
Number of racks: 1
FSCK ended at Mon Apr 18 15:20:49 PDT 2016 in 15 milliseconds

The filesystem under path '/user/allen' is HEALTHY
[allen@iclient0 ~]$ hdfs fsck -racks
Connecting to namenode via http://cmaster0:50070/fsck?ugi=all&racks=1&path=%2F
FSCK started by allen (auth:SIMPLE) from /192.168.170.138 for path / at Mon Apr 18 15:21:36 PDT 2016
.....Status: HEALTHY
Total size: 15445852 B
Total dirs: 91
Total files: 193
Total symlinks: 0
Total blocks (validated): 189 (avg. block size 81724 B)
Minimally replicated blocks: 189 (100.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 6 (3.1746032 %)
Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 3
Average block replication: 3.0317461
Corrupt blocks: 0
Missing replicas: 36 (5.91133 %)
Number of data-nodes: 4
Number of racks: 1
FSCK ended at Mon Apr 18 15:21:36 PDT 2016 in 131 milliseconds

The filesystem under path '/' is HEALTHY
[allen@iclient0 ~]$
```

* 检测是否有文件下是否有受侵蚀块

* 生成文件块下块报告

* 生成机架报告

图 4-34 fsck 选项及其输出

4.5 实战 WebHDFS

HDFS 自带 Web 界面主要用于显示文件系统和 HDFS 统计信息, 功能过于简单, 显然专为 HDFS 管理员开发。不过, HDFS 提供了功能强大的 WebHDFS REST API, 前端程序员可调用该 API 开发出复杂用户界面。

4.5.1 WebHDFS 简介

在 HDFS 中, WebHDFS 是一相对独立的模块, 历史上, 早期的 HDFS 里并没有这个模块, 随着 HDFS 应用领域越来越广, 大量 HDFS 使用者希望通过 HTTP 方式操作 HDFS, WebHDFS 随即应运而生。下面将主要围绕 WebHDFS 安全机制、传输协议和常用操作来讲述 WebHDFS。

1. 安全机制

作为 HDFS 一个独立模块, WebHDFS 直接采用了 HDFS 的安全措施, 而 HDFS 安全机制其实也就是 Hadoop 的安全机制, 即:

- 服务级安全措施
- 访问控制级安全措施

同理 WebHDFS 也有这两种安全机制, 由于这两层安全机制针对的对象并不相同, 故可以同时存在, 当使用 WebHDFS 时, 可以选择开启安全检测, 也可以不开启安全检测, 下面分别介绍。

(1) 未开启 security

默认安全机制, 又称 Simple 型 security。请注意, 未开启 security 不是指没有任何安全措施, 当未开启 security 时, WebHDFS 依旧有一层较弱的安全措施。具体到请求时, 就是 WebHDFS 会根据 URL 请求中的 `user.name` 参数确定用户, 若 URL 中未写明此参数, 则读取浏览器的默认用户名。

(2) 开启 security

此时认证由 Hadoop delegation token (授权令牌) 或 Kerberos SPNEGO 托管。即当请求 URL 中写明授权令牌时, HDFS 服务负责认证此 token (令牌), 若请求 URL 并未写明授权令牌, 则 Kerberos 认证服务器负责认证 URL 提交者的 Kerberos SPNEGO。

刚装好的 littleCstor 默认使用 Simple 型 security，此时只有一层最简单的安全措施，且此安全措施的核心是验证用户名，就是验证请求提交方的用户名。比如使用 Shell 提交请求时，认证 Shell 执行者用户名；同理，当请求由浏览器提交时，认证浏览器用户名。

相对于 Simple 型 security，Kerberos 安全机制认证非常苛刻，可以说它能够保障 99% 安全。但 Kerberos 本身相当复杂，最好能由工程师单独维护，为尽量保持简单，本书中 littleCstor 只采用 Simple 型安全机制，不过对中型以上的集群，建议开启 Kerberos。



图 4-35 WebHDFS API 合集

2. 数据提交方式

类似于普通 HTTP 请求，WebHDFS 支持 GET、PUT、POST 和 DELETE 这四大常见请求方式，图 4-35 罗列了这四种提交方式及其支持操作。

熟悉 Web 开发的读者应当知道，GET 是从服务器上获取数据，POST 则是向服务器发送数据，GET 支持的数据量较小，而 POST 支持的数据量大。当向 WebHDFS 发出请求，打开文件时，实际上打开一个句柄，并没有真实打开文件，故此操作可放在 GET 里请求。而 append 操作可能会裹挟大量数据写入远程文件，故 append 操作采用 POST 里方式，其他操作请读者自行分析。

4.5.2 WebHDFS 示例

通过 WebHDFS API 可以实现对 HDFS 增删改查操作, 下面以新建、查看等操作为例, 详细讲述 WebHDFS API。

1. 写文件

下面的操作完成在 excellent 机上调用 WebHDFS 接口, 实现在 HDFS 里新建文件 “/user/allen/love.txt”, 并将 excellent 机上文件 ilove.txt 里的内容写入 HDFS 的 love.txt 文件里。

既然要将 love.txt 内容上传, 那么 excellent 上就要存在这个文件, 请读者使用自己熟悉的编辑器在 excellent 机上新建 ilove.txt 并写入如下内容:

```
I love you njupt
I love you nanjing
I love you china
```

Step1 以 allen 身份 (user.name 参数) 向主节点申请新建文件/user/allen/love.txt。

```
[allen@excellent ~]$ curl -i -X PUT "http://cmaster0:50070/webhdfs/v1/user/allen/love.txt?user.name=all&op=CREATE"
```

Step2 使用 Step1 的返回地址, 执行新建并写入命令, 编者 step1 返回地址是:

```
http://cslave2:50075/webhdfs/v1/user/allen/love.txt?op=CREATE&user.name=all&namenoderpcaddress=cmaster0:8020&overwrite=false
```

故编者使用此返回地址, 在 excellent 上执行如下命令, 其中 -X 说明此 URL 采用 PUT 方式提交; -T 参数指定待上传文件 ilove.txt, 命令如下:

```
[allen@excellent ~]$ curl -i -X PUT -T ilove.txt "http://cslave2:50075/webhdfs/v1/user/allen/love.txt?op=CREATE&user.name=all&namenoderpcaddress=cmaster0:8020&overwrite=false"
```

需要注意的是, 由于 step1 的请求 URL 参数里并没指定覆盖属性, 故 Step2 执行前 (不是 Step1, Step1 只是申请), HDFS 的 “/user/allen” 目录下不可以存在文件 love.txt (图 4-36)。此外, 请求 URL 里必须写明 user.name=all, 否则会提示权限出错。

在图 4-37 中, 编者演示了完整操作过程, 从过程可以看出, Step2 需要使用 Step1 给出的地址。图 4-38 则为通过 Web 界面查看操作结果, 从图中可以看出, 通过 WebHDFS, 成功地在 HDFS 下新建了 love.txt 文件。

这里之所以要分两步完成创建文件和写入数据, 是因为申请操作都必须到主节点 cmaster0 去申请, 而实际写入数据时, 数据并不经过 cmaster0 (否则会导致集群瓶颈), 直接写入 cmaster0 指定的从节点。



图 4-36 确认 hdfs 里无 love.txt



图 4-37 WebHDFS 新建并写入内容操作过程

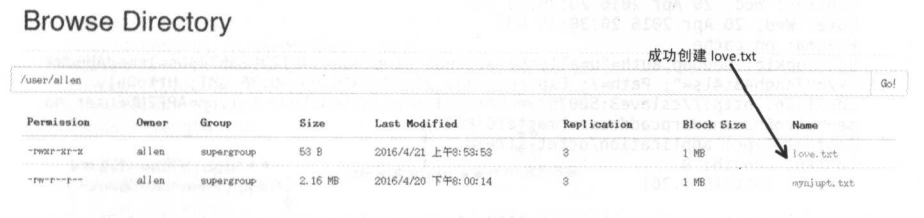


图 4-38 确认 WebHDFS 操作成功

2. 追加文件

下面的操作实现在 excellent 机上调用 WebHDFS API, 实现向 HDFS 中 (已存在) 文件“/user/allen/love.txt”里追加内容, 需要追加的内容在 excellent 机上文件 iloveApd.txt 里。

既然需要用到 `iloveApd.txt`, 那就需要在 `excellent` 上存在这个文件, 故在 `excellent` 上先创建文件 `iloveApd.txt` 并写入如下内容:

```
dear dad and mom
dear brothers and sisters
```

接着, 依旧分为 Step1 和 Step2 两个步骤, 操作 WebHDFS API, 操作内容如下:

Step1 以 `allen` 身份 (`user.name` 参数设定) 向主节点 `cmaster0` 提出申请, 申请向文件 `"/user/allen/love.txt"` 里追加内容。

```
[allen@excellent ~]$ curl -i -X POST "http://cmaster0:50070/webhdfs/v1/user/allen/love.txt?user.name=allen&op=APPEND"
```

Step2 使用 Step1 的返回地址, 执行追加命令, 编者 step1 返回地址是:

```
http://cslave3:50075/webhdfs/v1/user/allen/love.txt?op=APPEND&user.name=allen&namenoderpcaddress=cmaster0
```

故编者使用此返回地址, 在 `excellent` 上执行如下命令, 其中 `-X` 说明此 URL 采用 POST 方式提交; `-T` 参数指定待上传文件 `iloveApd.txt`, 命令如下:

```
[allen@excellent ~]$ curl -i -X POST -T iloveApd.txt "http://cslave3:50075/webhdfs/v1/user/allen/love.txt?op=APPEND&user.name=allen&namenoderpcaddress=cmaster0:8020"
```

图 4-39 为向 `love.txt` 文件追加内容的整个执行过程。读者应当注意到, 这两个例题都是使用 Curl 提交 WebHDFS 请求, 不同的是, 若请求操作为 APPEND, 则 Curl 声明的提交方式为 POST, 若请求操作为 CREATE, 则提交方式为 PUT, 这正与图 4-37、图 4-39 中所述相对应。

```
[allen@excellent ~]$ cat iloveApd.txt ← ** 确定本地存在 ilove.txt
dear dad and mom
dear brothers and sisters
[allen@excellent ~]$ curl -i -X POST "http://cmaster0:50070/webhdfs/v1/user/all
en/love.txt?user.name=allen&op=APPEND"
HTTP/1.1 307 TEMPORARY_REDIRECT
Cache-Control: no-cache
Expires: Wed, 20 Apr 2016 20:30:35 GMT
Date: Wed, 20 Apr 2016 20:30:35 GMT
Pragma: no-cache
Expires: Wed, 20 Apr 2016 20:30:35 GMT
Date: Wed, 20 Apr 2016 20:30:35 GMT
Pragma: no-cache
Set-Cookie: hadoop.auth="u=allen&p=allen&t=simple&e=1461220235744&s=1omedAH6dW4
bk2mV7shghA3t4ls="; Path=/; Expires=???, 21-??-2016 06:30:35 GMT; HttpOnly
Location: http://cslave3:50075/webhdfs/v1/user/allen/love.txt?op=APPEND&user.na
me=allen&namenoderpcaddress=cmaster0:8020
Content-Type: application/octet-stream
Content-Length: 0
Server: Jetty(6.1.26)
↑ ** Step1: 申请
↑ ** Step1的答复地址
** Step2: 使用Step1答复地址
将本地文件iloveApd.txt追加至HDFS
[allen@excellent ~]$ curl -i -X POST -T iloveApd.txt "http://cslave3:50075/webh
dfs/v1/user/allen/love.txt?op=APPEND&user.name=allen&namenoderpcaddress=cmaster
0:8020"
HTTP/1.1 100 Continue
↑ ** 客户端收到
追加成功协议 200
HTTP/1.1 200 OK
Content-Length: 0
Connection: close
[allen@excellent ~]$
```

图 4-39 WebHDFS 文件追加操作

3. 查看文件

下面的操作完成使用 brilliant 机的 FireFox，excellent 机命令行，调用 WebHDFS，查看 HDFS 上文件“/user/allen/love.txt”。



图 4-40 WebHDFS 服务 open 操作示例

在 brilliant 机的 FireFox 地址栏输入如下 URL，其执行结果如图 4-40 所示。
`http://cmaster0:50070/webhdfs/v1/user/allen/love.txt?op=OPEN`

同理，在 excellent 机的命令行下输入如下命令，其执行过程和结果如图 4-41 所示：
`[allen@excellent ~]$ curl -i -L "http://cmaster0:50070/webhdfs/v1/user/allen/love.txt?op=OPEN"`

这两种访问方式本质上是相同的，因为 curl 就是利用 URL 语法在命令行下工作的文件传输工具，编者两者都列举的目的是让读者清晰明白 WebHDFS 的作用是定制 HDFS 读写页面。

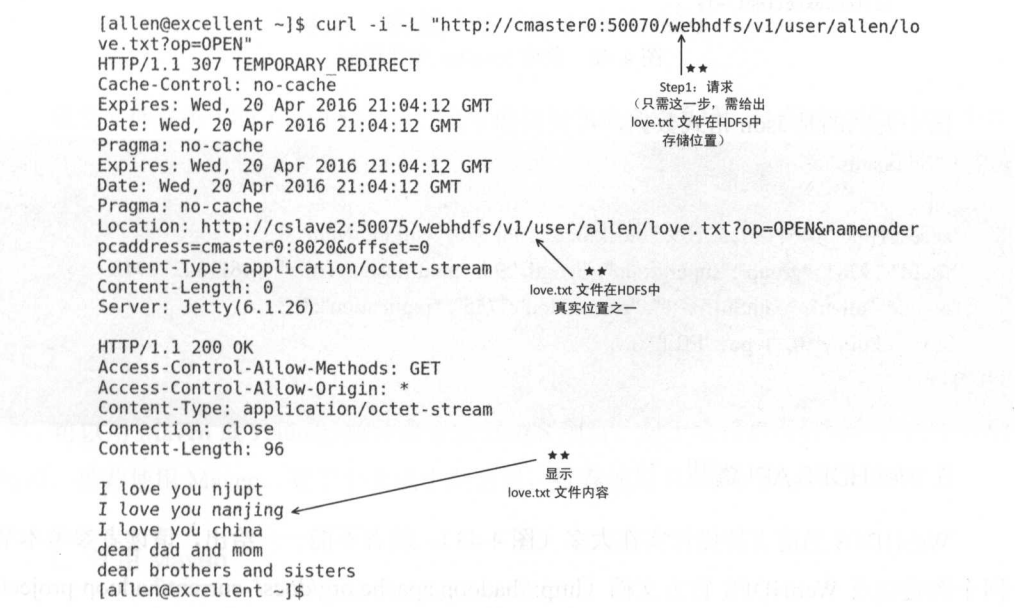


图 4-41 WebHDFS 服务 open 操作示例

4. 查看文件句柄

下面的操作完成在 excellent 上调用 WebHDFS，查看文件 “/user/allen/love.txt” 的句柄信息。

在 excellent 机上执行如下命令，命令中 op 参数说明此次请求获取 love.txt 文件句柄信息。

```
[allen@excellent ~]$ curl -i "http://cmaster0:50070/webhdfs/v1/user/allen/love.txt?op=GETFILESTATUS"
```

图 4-42 为命令执行过程，此外命令中并未写明 user.name=allan，这是因为从权限角度分析，该文件的可读权限面向所有用户。

```
[allen@excellent ~]$ curl -i "http://cmaster0:50070/webhdfs/v1/user/allen/love.txt?op=GETFILESTATUS"
HTTP/1.1 200 OK
Cache-Control: no-cache
Expires: Wed, 20 Apr 2016 21:13:04 GMT
Date: Wed, 20 Apr 2016 21:13:04 GMT
Pragma: no-cache
Expires: Wed, 20 Apr 2016 21:13:04 GMT
Date: Wed, 20 Apr 2016 21:13:04 GMT
Pragma: no-cache
Content-Type: application/json
Transfer-Encoding: chunked
Server: Jetty(6.1.26)

{"FileStatus":{"accessTime":1461186252105,"blockSize":1048576,"childrenNum":0,"fileId":19701,"group":"supergroup","length":96,"modificationTime":1461184279942,"owner":"allen","pathSuffix":"","permission":"755","replication":3,"storagePolicy":0,"type":"FILE"}}
[allen@excellent ~]$
```

执行成功代码200

只需一步
申请获取 love.txt
文件句柄信息

json格式love.txt
文件句柄信息

图 4-42 获取 love.txt 文件句柄

图中返回的是 Json 格式数据，可调整如下：

```
{ "FileStatus":
{
"accessTime":1461186252105,"blockSize":1048576,"childrenNum":0,
"fileId":19701,"group":"supergroup","length":96,"modificationTime": 1461184279942,
"owner":"allen","pathSuffix":"","permission":"755","replication":3,
"storagePolicy":0,"type":"FILE",...
}
}
```

5. WebHDFS API 集^[1]

WebHDFS 当前支持操作实在太多（图 4-43），编者不能一一给出，请读者参考本节四个例题以及 WebHDFS 官方文档（<http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/WebHDFS.html>），自行编写示例代码。

比如，读者可使用 Spring 开发工具包、WebHDFS API 接口，开发一套自定义 HDFS Web 页面，要求实现从页面上读写 HDFS 上 “/user/webuser” 目录下文件内容。

- Authentication
- Proxy Users
- File and Directory Operations
 - Create and Write to a File
 - Append to a File
- Other File System Operations
 - Get Content Summary of a Directory
 - Get File Checksum
- Extended Attributes(XAttr)s Operations
 - Set XAttr
 - Remove XAttr
- Snapshot Operations
 - Create Snapshot
 - Delete Snapshot
- Delegation Token Operations
 - Get Delegation Token
 - Get Delegation Tokens
- Error Responses
 - HTTP Response Codes
 - Illegal Argument Exception
 - Security Exception
- JSON Schemas
 - ACL Status JSON Schema
 - XAttr's JSON Schema
- HTTP Query Parameter Dictionary
 - ACL Spec
 - XAttr Name

限于篇幅
大量省略

WebHDFS支持
的全部操作

图 4-43 WebHDFSAPI 集

4.6 实战 HDFS JAVA API

磁盘文件系统（ETX、NFS）、内存文件系统、网络文件系统、分布式文件系统，不管文件系统有多少个分类，其基本功能就是存储文件。作为一种典型的分布式文件系统，HDFS 支持常规的读写操作。用户可以使用 JAVA API 对 HDFS 上的文件进行“增删改查”操作。

4.6.1 搭建开发环境

可以用 Maven 或 Hadoop 插件来开发 HDFS 项目，对于项目管理较为严格的中大型公司，推荐使用 Maven，对于个人或小型公司，推荐使用 Hadoop-plugin。

1. 使用 Maven

下载并解压 Eclipse，接着依次单击“Help→Eclipse Marketplace”；然后在新出现的对话框中，找到 Find 输入框并输入“maven”，单击“Go”；最后，选中和本 Eclipse 版本配套的 Maven，单击“Install”。图 4-44 为编者在 excellent 机上配置 Eclipse Maven 开发环境示意图。

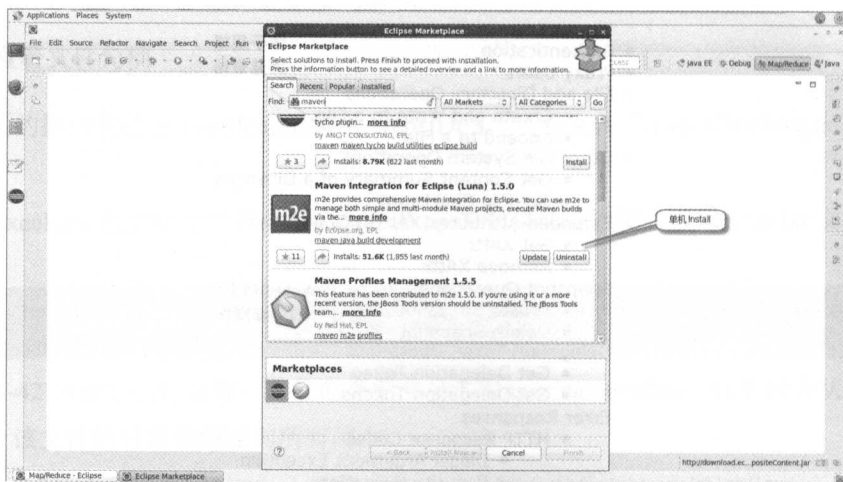


图 4-44 配置在 Eclipse 上安装 Maven 插件

使用该 Eclipse，依次单击“File→New→Other→Maven→Maven Project”，写入项目名称，并在项目 pom.xml 中建立如下依赖：

```
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-common</artifactId>
  <version>2.7.1</version>
</dependency>
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-hdfs</artifactId>
  <version>2.7.1</version>
</dependency>
<dependency>
  <groupId>org.apache.hadoop</groupId>
  <artifactId>hadoop-client</artifactId>
  <version>2.7.1</version>
</dependency>
```

最后，新建 Class 并编写用户代码即可，图 4-45 为编者使用 Maven 新建 bg-hdfs 项目示意图。

由于部分组件网速较慢，使用 Maven 会让项目管理变得很复杂，影响工作效率，对于个人或小型项目组，建议不使用 Maven。

2. 使用 Hadoop 插件

请通过互联网下载“hadoop-eclipse-plugin-2.X.0.jar”，接着将此插件拷贝至 Eclipse 安装目录下的“plugin”文件里。新建时，可直接新建 map/reduce 项目。请读者使用搜索引擎，搜索 Hadoop Eclipse 插件，网上的图文教程非常详细。

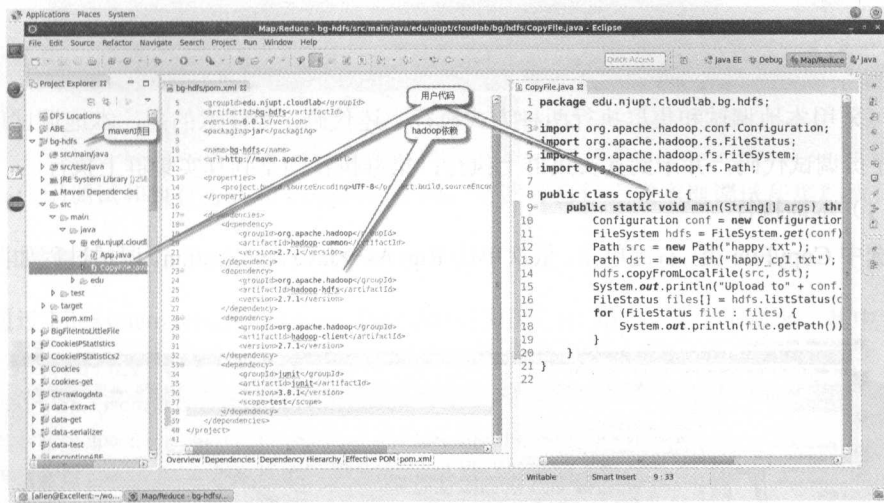


图 4-45 使用 Maven 管理 HDFS 项目

4.6.2 常规操作示例

本节将提供几个简单的实例，指导读者如何使用 Hadoop 的 Java API 针对 HDFS 进行文件上传、创建、重命名、删除等操作。

1. 上传本地文件到 HDFS

通过 `FileSystem.copyFromLocalFile(Path src, Path dst)` 可将本地文件上传到 HDFS 的指定位置上，其中 `src` 和 `dst` 均为文件的完整路径。具体实例如下：

```
package edu.njupt;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
public class CopyFile {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        FileSystem hdfs = FileSystem.get(conf);
        Path src = new Path("happy.txt");
        Path dst = new Path("happy_cp1.txt");
        hdfs.copyFromLocalFile(src, dst);
        System.out.println("Upload to" + conf.get("fs.default.name"));
        FileStatus files[] = hdfs.listStatus(dst);
        for (FileStatus file : files) {
            System.out.println(file.getPath());
        }
    }
}
```



可采用本地调试和集群执行两种方式执行上述代码，要注意的是，本地调试仅为方便程序员调试代码，并未到真实集群上执行；集群执行则是到真实集群上执行。

(1) 本地执行

选中 CopyFile 类，右键单击，依次单击 Run As→Java Application。调试过程如图 4-46 所示。

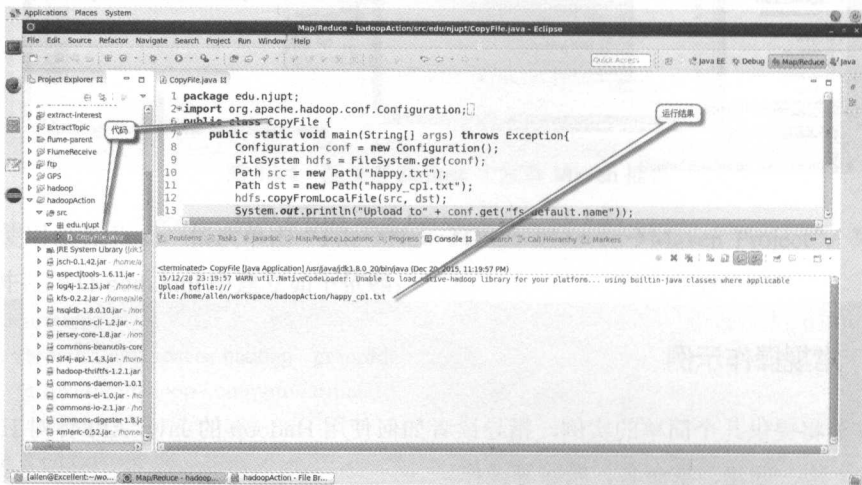


图 4-46 本地调试 HDFS 项目

(2) 集群执行

将上述代码打包后，上传至 iclient0 机，使用下述命令执行即可：

```
[allen@iclient0 ~]$ hadoop jar ~/hdpAction.jar edu.njupt.CopyFile
```

由于 CopyFile 类里无需入参，故上述命令无须其他参数，直接执行即可。

2. 新建文件

通过 FileSystem.create(Path f)可在 HDFS 上创建文件，其中 f 为文件的完整路径。具体实现如下：

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
public class CreateFile {
    public static void main(String[] args)throws Exception{
        Configuration conf=new Configuration();
        byte[] buff ="hello word!".getBytes();
        FileSystem hdfs = FileSystem.get(conf);
```

```

    Path dfs = new Path("test");
    FSDataOutputStream outputStream = hdfs.create(dfs);
    outputStream.write(buff, 0, buff.length);
}
}

```

该代码调试和执行方式和 CopyFile 相同，请读者自行完成本地调试和集群执行。

3. 重命名

通过 `FileSystem.rename(Path src, Path dst)` 可为指定的 HDFS 文件重命名，其中 `src` 和 `dst` 均为文件的完整路径。具体实现如下：

```

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
public class Rename {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        FileSystem hdfs = FileSystem.get(conf);
        Path fspath = new Path("test");
        Path topath = new Path("test-mv");
        boolean isRename = hdfs.rename(fspath, topath);
    }
}

```

该代码调试和执行方式和 CopyFile 相同，请读者自行完成本地调试和集群执行。

4. 查看 HDFS 文件的最后修改时间

通过 `FileStatus.getModificationTime()` 可查看指定 HDFS 文件的修改时间。具体实现如下：

```

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
public class GetLTime {
    public void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        FileSystem hdfs = FileSystem.get(conf);
        Path fpath = new Path("test-mv");
        FileStatus fileStatus = hdfs.getFileStatus(fpath);
        long modificationTime = fileStatus.getModificationTime();
        System.out.println("Modification time is: " + modificationTime);
    }
}

```

该代码调试和执行方式和 CopyFile 相同，请读者自行完成本地调试和集群执行。

5. 查看某个 HDFS 文件是否存在

通过 `FileSystem.exists(Path f)` 可查看指定 HDFS 文件是否存在, 其中 `f` 为文件的完整路径。具体实现如下:

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
public class CheckFile {
    public void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        FileSystem hdfs = FileSystem.get(conf);
        Path findf = new Path("test-mv");
        boolean isExists = hdfs.exists(findf);
        System.out.println("Exist?" + isExists);
    }
}
```

该代码调试和执行方式和 `CopyFile` 相同, 请读者自行完成本地调试和集群执行。

6. 查看某个文件在 HDFS 集群的位置

通过 `FileSystem.getFileBlockLocations(FileStatus file, long start, long len)` 可查找指定文件在 HDFS 集群上的位置, 其中 `file` 为文件的完整路径, `start` 和 `len` 来标识查找文件的路径。具体实现如下:

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.BlockLocation;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
public class FileLoc {
    public void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        FileSystem hdfs = FileSystem.get(conf);
        Path fpath = new Path("test-mv");
        FileStatus filestatus = hdfs.getFileStatus(fpath);
        BlockLocation[] blkLocations = hdfs.getFileBlockLocations(filestatus, 0, filestatus.getLen());
        int blockLen = blkLocations.length;
        for (int i = 0; i < blockLen; i++) {
            String[] hosts = blkLocations[i].getHosts();
            System.out.println("block" + i + "location:" + hosts[i]);
        }
    }
}
```

该代码调试和执行方式和 `CopyFile` 相同, 请读者自行完成本地调试和集群执行。

7. 删除 HDFS 上的文件

通过 `FileSystem.delete(Path f, Boolean recursive)` 可删除指定的 HDFS 文件，其中 `f` 为需要删除文件的完整路径，`recursive` 用来确定是否进行递归删除。具体实现如下：

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
public class DeleteFile {
public static void main(String[] arg)throws Exception{
    Configuration conf=new Configuration();
    FileSystem hdfs=FileSystem.get(conf);
    Path delef=new Path("test-mv");
    boolean isDeleted=hdfs.delete(delef,false);
    //递归删除
    //boolean isDelete=hdfs.delete(delef,true)
    System.out.println("delete?" + isDeleted);
    }
}
```

该代码调试和执行方式和 `CopyFile` 相同，请读者自行完成本地调试和集群执行。

8. 获取 HDFS 集群上所有节点名称

通过 `DatanodeInfo.getHostName()` 可获取 HDFS 集群上的所有节点名称。具体实现如下：

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.hdfs.DistributedFileSystem;
import org.apache.hadoop.hdfs.protocol.DatanodeInfo;
public class GetList {
    public void mian(String[] args)throws Exception{
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(conf);
        DistributedFileSystem hdfs = (DistributedFileSystem) fs;
        DatanodeInfo[] dataNodeStats = hdfs.getDataNodeStats();
        String[] names = new String[dataNodeStats.length];
        for (int i = 0; i < dataNodeStats.length; i++) {
            names[i] = dataNodeStats[i].getHostName();
            System.out.println("node"+i+"name"+ names[i]);
        }
    }
}
```

4.7 实战 HDFS 大项目：用 HDFS 存储海量视频数据

4.7.1 应用场景

随着时代的发展，高清视频的应用日益广泛。与此同时，高清视频监控项目规模也在不断扩大，因此，高清视频的存储越来越受到人们的关注。对于视频监控而言，图像清晰度无疑是最关键的特性。图像越清晰，细节便越明显，观看体验越好，各种智能应用业务的准确度也越高。然而，高清的视频数据以 G 为级别大小也是不可小觑的。与此同时，面对如潮水般涌现的海量视频数据，不仅对存储容量有很高的要求，对读写性能、可靠性等都提出了很高要求。因此，选择什么样的存储系统，往往成为影响视频读写速度的关键。

Hadoop 的 HDFS 具有扩展性强、可靠性高、成本低等优势，为高清视频的存储提供了很大的便利。其中，HDFS “一次写入多次读取” 的文件访问模型简化了视频流数据一致性问题，并且保证了高吞吐量的数据访问。另外，HDFS 的多机架存放副本的策略使用户不用担心因为某个 DataNode 的故障而导致的视频文件不完整，确保高清视频实时可用。

4.7.2 设计实现

1. 模拟视频流

在缺少摄像头的情况下可以使用 VLC 播放器模拟出 H264 的实时视频流。

(1) 搭建组播服务器

Step1 运行 VLC 程序后选择“媒体→串流”。

Step2 通过“添加”选择需要播放的文件（以 WMV 文件为例），单击“串流”。

Step3 流输出有三项需要设置，包括来源、目标和选项。来源刚才已指定，单击“下一个”。

Step4 勾选“在本地显示”，选择“RTP/MPEG Transport Stream”输出，单击“添加”。

Step5 如果建立 IPv6 组播服务器，可以输入组播地址 ff15::1，指定端口号“5004”，单击右下角的“下一个”。如果需要建立 IPv4 组播服务器，则地址栏可输入“239.1.1.1”（239.0.0.0/8 为本地管理组播地址）。

Step6 将 TTL 设置为 10，单击左下角“串流”即可发送组播视频，同时在本地播放（视频打开时间较慢，需要等待半分钟左右）。

Step7 使用 WireShark 抓包查看。

(2) 搭建组播客户端

Step1 运行程序后选择“媒体→打开网络串流”。

Step2 输入 URL (rtp://@[ff15::1]:5004)，单击“播放”就可以观看组播视频，如果为 IPv4 组播环境，可输入 rtp://239.1.1.1:5004。

注意，测试前请关闭 PC 防火墙，以免影响组播报文的发送和接收。

2. 接收视频流

由于本实战部分的重点为海量视频数据存储，因此不再展示接收视频流的代码，读者可从 <http://bbs.chinacloud.cn/> 下载。本处采用的接收方法为：将从摄像头接收到的或通过模拟产生的视频流，以文件的形式存储在本地文件夹内。此外，这个过程不产生任何中间文件。

3. 海量视频数据存储

存储海量视频数据的思路为：通过 Hadoop 提供的 API 接口，实现将接收到的视频流文件从本地上传到 HDFS 中。

在此过程中，接收到的视频文件将源源不断地存储到一个指定的本机文件夹中，因此，这个本地文件夹的文件是在动态增加的，此处将这个动态变化的文件夹当做一个“缓冲区”，然后以流的形式将“缓冲区”文件和 HDFS 进行对接，之后通过调用 `FSDDataOutputStream.write(buffer, 0, bytesRead)` 实现以流的方式将本地文件上传至 HDFS 上。当本地文件上传成功后，再调用 `File.delete()` 批量删除“缓冲区”中已上传文件。此过程将一直延续，直到所有文件都上传至 HDFS 且清空本地文件夹后才结束。

将接收到的视频流文件上传到 HDFS 的完整代码如下：

```
import java.io.File;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDDataInputStream;
import org.apache.hadoop.fs.FSDDataOutputStream;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
public class Read2 {
    public static void main(String [] args)throws Exception{
        Configuration conf=new Configuration();
        FileSystem hdfs=FileSystem.get(conf);
        FileSystem local=FileSystem.getLocal(conf);
        //确定需要上传视频流路径和接受视频流路径
```

```

Path inputDir=new Path("C: \\msys\\1.0\\home\\admin\\ffmpeg\\live.264");
Path hdfsFile=new Path("/acceptFile/");
//System.out.println(inputDir.toString());
//创建 HDFS 上 "acceptFile" 目录, 用以接收视频文件
hdfs.mkdirs(hdfsFile);
FileStatus[] inputFiles=local.listStatus(inputDir);
FSDataOutputStream out;
//通过 OutputStream.write( )来将视频文件循环写入 HDFS 下的指定目录
for(int i=0;i<inputFiles.length;i++){
    System.out.println(inputFiles[i].getPath().getName());
    FSDataInputStream in=local.open(inputFiles[i].getPath());
    out=hdfs.create(new Path("/acceptFile /"+inputFiles[i].getPath().getName()));
    byte buffer[]=new byte[256];
    int bytesRead=0;
    while((bytesRead=in.read(buffer))>0){
        out.write(buffer,0,bytesRead);
    }
    out.close();
    in.close();
    File file = new File(inputFiles[i].getPath().toString());
    file.delete();
}
}
}

```

习 题

1. 简述海量数据造成了哪些存储问题。
2. 简述 HDFS 功能作用及其体系架构。
3. 简述 HDFS 采用何种机制确保数据安全、可靠。
4. 简述手工部署 HDFS、使用 Ambari 部署 HDFS 的部署步骤。
5. 简述 HDFS 访问接口。
6. 简述使用 Maven 时, HDFS 开发环境搭建步骤, 不使用 Maven 时又如何?
7. 请分别使用 Text、Sequence、Compress 格式和自定义格式读写 HDFS 文件。
8. 当 HDFS 块大小调整后, 集群中已存数据块大小是否相应变化。
9. 试着开启 HDFS 快照机制, 压缩机制。
10. 在大型系统中, 如何使用 HDFS 来实现数据持久化?

11. 不使用 MapReduce 时, 如何快速定位 HDFS 中的文件 (提示 MapFile) ?
12. 简述 HDFS 的数据和服务安全机制。

参考文献

- [1] <http://datascienceseries.com/blog/digital-universe-will-grow-to-40zb-in-2020-with-a-62-share-for-emerging-markets>
- [2] [https://en.wikipedia.org/wiki/Master/slave_\(technology\)](https://en.wikipedia.org/wiki/Master/slave_(technology))
- [3] <https://en.wikipedia.org/wiki/RAID>
- [4] <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html>
- [5] <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html>
- [6] <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html>
- [7] [https://en.wikipedia.org/wiki/Split-brain_\(computing\)](https://en.wikipedia.org/wiki/Split-brain_(computing))
- [8] <http://doc.mapr.com/display/MapR/Home>
- [9] <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html>
- [10] <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/LibHdfs.html>
- [11] <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/WebHDFS.html>

第5章 分布式资源管理器 YARN

HADOOP
BEING DIGITAL

我们知道,操作系统是一个用于管理计算机硬件资源并提供用户接口的一套系统软件。从这个概念上看(无论是功能还是定位),YARN 都像一个分布式操作系统(严格称为分布式资源管理器)。本章将首先讲述操作系统的概念并由此引入 YARN,接着重点讲解 YARN 的体系架构。在最后实战环节,编者将会在 littleCstor 实际操作 YARN Shell 和 YARN 编程。

5.1 分布式资源管理器引例

一个中小型互联网公司一般拥有 20 台左右线下服务器,这些线下服务器大多用于离线数据分析、新产品开发、测试等。以离线分析为例,起初数据量较小时,单机程序即可完成分析任务。随着业务发展,公司需要分析的数据已超过单机处理能力,这时工程师可通过封装 Netty、Thrift 等第三方工具,实现单机程序互连。可是随着业务进一步发展,当工程师试图将小集群上的程序进一步扩展时,由于分布式环境的复杂性,此类程序已很难实现再扩展。

YARN 即是这样一套基础分布式架构,其直接摒弃程序的业务逻辑,只保留了资源管理、仲裁和作业调度功能(程序业务逻辑模块直接下放到 Yarn-App 里)。这点(资源管理、仲裁、作业管理)和操作系统中内存管理、进程管理这两个功能类似,故下面编者以操作系统为切入点,引出 YARN。

5.1.1 分布式资源管理器简介

实际上 YARN 是单机操作系统的分布式发展,下面先回顾单机操作系统的主要功能,然后再讲述分布式操作系统应当具备的功能。

1. 单机操作系统

操作系统(Operating Systems, OS)^[1]是介于硬件裸机和用户之间的一层系统软件。从用户的角度来看,OS 是用户与计算机硬件系统之间的接口;从资源管理角度来看,OS 是计算机系统资源的管理者。总之,OS 实现了对计算机资源的抽象,隐藏了对硬件操作的细节,使用户能更方便地使用计算机。

图 5-1 是单机操作系统的示意图,从图中可以看出,向下,OS 为用户和应用程

序屏蔽了计算机硬件细节；向上，OS 高效地管理着各种计算机硬件资源，统一并协调了用户和应用程序对硬件的相关操作。读者须注意，实际上“用户本身”就是一种特殊的应用程序，同样，应用程序也可以看成一种特殊的用户。

进一步，当前单机 OS 的主要功能为：处理器管理/进程管理、存储器管理、设备管理、文件管理、操作系统与用户之间的接口。

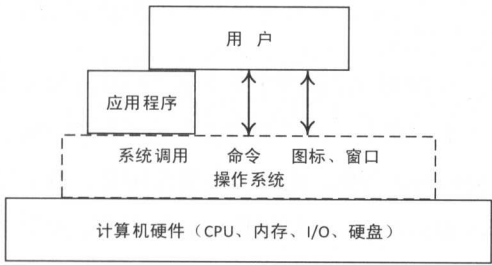


图 5-1 OS 作用接口的示意图

其中，前四个功能是“管理计算机资源”的细化，后一个功能“操作系统与用户之间的接口”是“硬件和用户之间的桥梁”的具体体现。

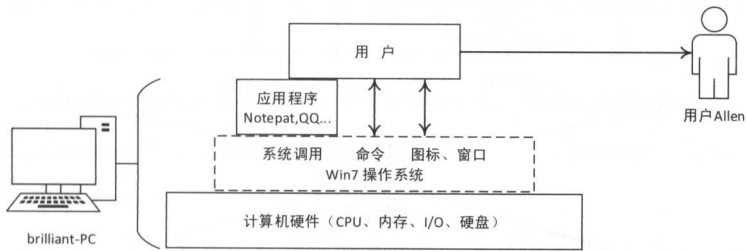


图 5-2 aiAllen.txt 文件操作示意图

以 Windows 7 为例，用户 Allen 登录 brilliant 并新建一文档 aiAllen.txt，此时 OS 会启动进程 notepad.exe。向上，notepad.exe 进程负责和用户 Allen 交互；向下，notepad 通过系统调用（或直接）和 OS 交互，而 OS 则会直接（通过驱动）操控硬件。比如 Allen 发出保存操作，此时 notepad.exe 会调用 OS 存储命令，接着这些存储命令被 OS 转化为一系列有序 I/O 指令，最后硬盘驱动程序根据这些指令进行最终保存操作（图 5-2）。

在桌面上新建文件，实际上就是在使用 OS 的窗体接口功能。OS 启动 notepad.exe 是进程管理；Allen 向 aiAllen.txt 里输入内容实际上是通过 notepad.exe 向特定内存区域写数据，这必然涉及内存管理；新建文件名为 aiAllen.txt，notepad.exe 会按文件管理协议为此文件新建文件控制块 FCB，这又涉及文件管理。最后，文件保存涉及 I/O 设备管理。

2. 分布式操作系统

分布式操作系统（Distributed Operating Systems, DOS）是从单机 OS 发展而来的，其主要用于管理集群资源和提供用户接口。从用户的角度来看，DOS 是用户与计算机硬件系统之间的接口；从资源管理角度来看，DOS 又是集群中所有计算机系统资源的管理者。DOS 抽象了集群中每台机器上最重要的几类资源（CPU、内存、存储、网络），使上层程序能更方便地使用这些资源。

图 5-3 就是这样的—个 DOS 示例，图中共有 N 台 slave 机和 1 台 master 机，假定“总管进程”部署在 cmaster1 上，向上，该“管理者”为上层的 MR、Spark、Hive 等应用屏蔽了下层复杂结构，以 MR-App 为例，此时其主要作用是，当上层用户提交 MR-App 时，该“管理者”负责接收该 MR-App，并将集群资源按需分配给该 MR-App；向下，“总管进程”统一管理着集群内所有资源。

综上，分布式操作系统主要作用为：管理分布式系统资源、控制分布式程序运行和提供用户（包括程序用户和人）接口。

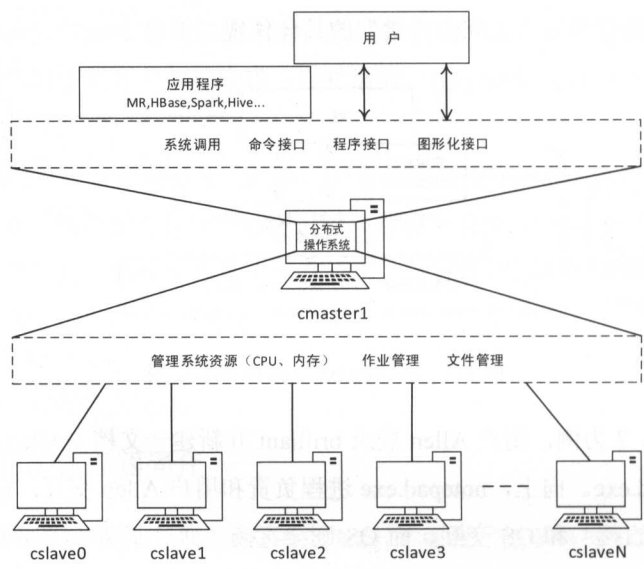


图 5-3 分布式操作系统所处层次位置示意图

本将所讲述的 YARN 就是分布式操作系统的一个典型特例，在整个大数据组件中，YARN 的定位是：

- 集群资源管理中心
- 任务调度中心

这两个功能正是单机 OS 的核心模块，当 YARN 运行时，其须不停汇总集群内各机资源（CPU、内存、网络），当用户提交 Yarn-App 时，YARN 须按照预先设定的调度策

略为该 Yarn-App 分配资源。

有了上述角色定位，可将 YARN 功能进一步细化为：

- 任务调度
- 资源管理
- 用户接口

和单机 OS 相比，YARN 好像少了文件管理和设备管理，这是在分布式系统中，文件管理已由 HDFS 负责，加之集群中并不存在设备这一概念，故 YARN 主要任务是管理运行在 YARN 上的不同 Yarn-App，管理集群内所有资源和为用户提供 Web、命令行等接口。

5.1.2 分布式资源管理器架构

通过上述内容知道了 YARN 的角色定位和功能模块，现在的问题是如何构建 YARN？我们知道典型的分布式系统皆采用 master/slave 架构，slave 进程负责监管本机资源，master 进程负责汇总各个 slave 上报的资源信息。故下面也采用 master/slave 架构来构建一个分布式操作系统（不妨称 preYARN）。

在构建 preYARN 之前，为了方便资源管理与分配，现将一定量的各类资源集抽象成一个逻辑概念 Container^[2]。比如统一规定<1 核,2G>为一个 Container，又比如还可规定<2 核，4G>为一个 Container。

显然，不管规则如何变化，只要制定了标准，划分结果就一定相同，比如 master 进程规定所有机器都采用一个标准，将<1 核,2G>视为一个 Container，那么不管集群中有多少机器，单机划分 Container 的结果总是相同的，Container 的概念能够大大简化 preYARN 资源管理模块。

现有一些配置完全相同的机器 cmaster0~cmaster2、cslave0~cslaveN（图 5-4），已知每台机器都是 1 个双核 CPU，4G 内存，下面就使用上述机器构建 preYARN。

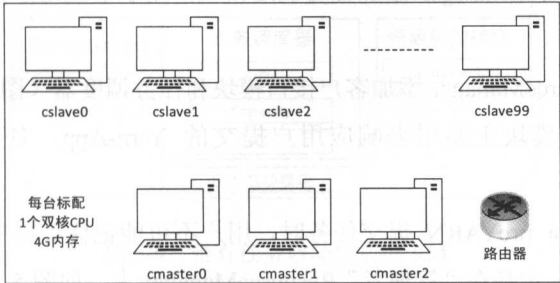


图 5-4 场景硬件配置图

Step1 构建从进程 NodeManager，用于管理本机资源。

Step2 构建主进程 ResourceManager，用于汇总各个 slave 进程上传的单机资源信息。

现规定，上述的 NodeManager 和 ResourceManager 采用 master/slave 架构，ResourceManager 为 master，NodeManager 为 slave。此外，每个 slave 机上部署一个 NodeManager 进程，该进程须不断更新本机资源状态信息并定期向 ResourceManager 上传这些信息。而 ResourceManager 则接收各 NodeManager 上传的资源信息，然后将这些信息汇总成一张完整资源表（图 5-5）。

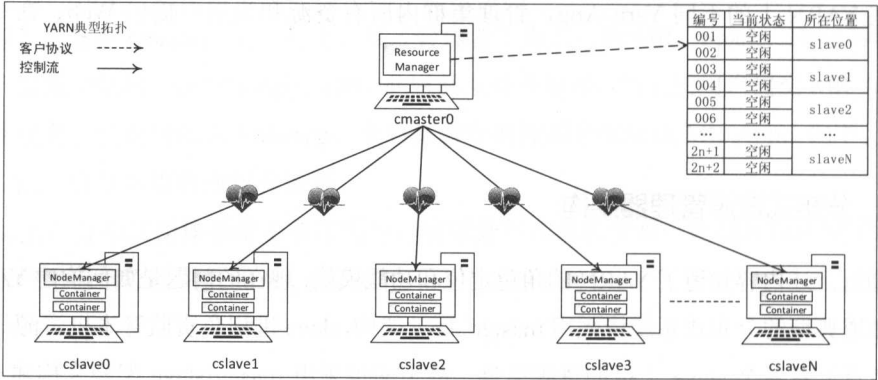


图 5-5 YARN 体系架构图

注意图 5-5 中的 Container 为一个逻辑概念，其实际并不存在，比如当规定 <1 核, 2G> 为一个 Container 时，则各 slave 都持有两个 Container。引入 Container 这一逻辑概念后，就有了参考的尺度，能够大大方便 NodeManager 管理本机资源，简化 ResourceManager 管理集群资源。

显然经过 Step1 和 Step2 构建之后的 preYARN 仅有管理集群资源的能力，已无其他功能（图 5-6）。人类制造工具的目的是为了使用工具，同样现在制造了 preYARN 就要使用它，而想要使用 preYARN，则必然需要为 preYARN 添加一个任务响应模块（专业术语称调度器）。

下面就为 ResourceManager 添加任务调度模块，用于响应用户任务并为用户任务分配资源。

Step3 为 ResourceManager 添加客户接口模块和任务调度器（图 5-7）。

其中客户接口模块主要用来响应用户提交的 Yarn-App，任务调度器则用于为 Yarn-App 分配资源。

显然，当用户向 preYARN 提交任务时，用户不可能记住集群中的每一台机器，故 Yarn-App 响应模块一定是在“管理者”ResourceManager 上。如图 5-8 所示，当客户端向 preYARN 提交任务时，其首先向 ResourceManager 提交任务。故须在 RM 里添加“客户

接口模块”和“任务调度器”两个模块（图 5-7），其中“客户接口模块”专门用来接收 iclient0 提交的任务，“任务调度器”默认专门为 Yarn-App 分配资源（以 Container 形式）。

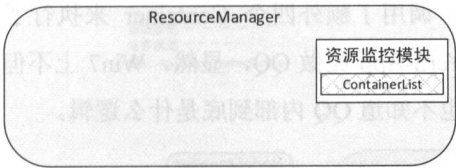


图 5-6 ResourceManager 功能模块



图 5-7 带有任务调度器的 RM

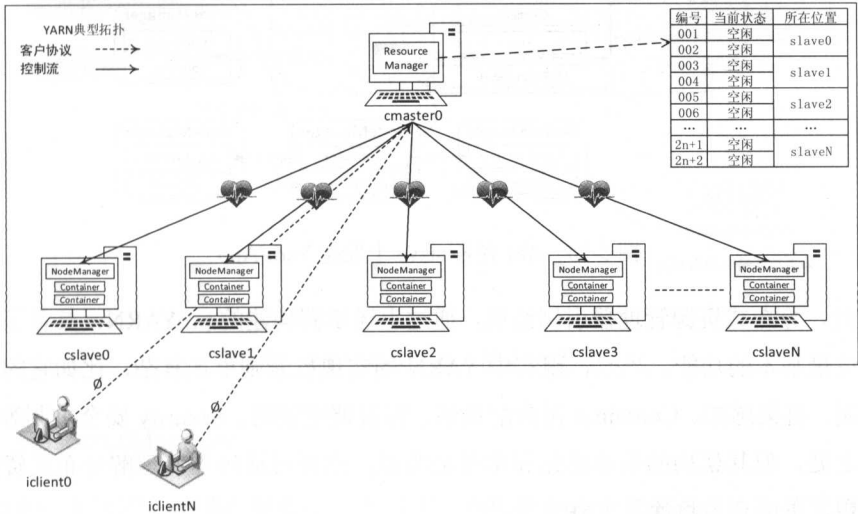


图 5-8 带有客户端的 preYARN

然而，当“任务调度器”为 iclient0 提交的 Yarn-App 分配资源后，并不代表着该任务就已经在执行了。以 Linux 操作系统为例，就绪状态的进程要被进程调度器选中后，才能执行。同样，当 Yarn-App 取得资源后，还要有一个单独模块用于启动该 Yarn-App（图 5-9），可描述为如下步骤。

Step4 为 ResourceManager 添加“任务启动模块”，负责在 Container 上启动 Yarn-App。

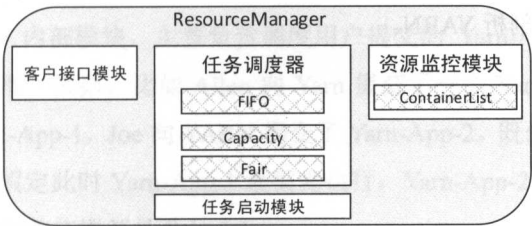


图 5-9 带有任务启动模块的 RM

待该模块将 Yarn-App 在某 Container 上拉起之后，至于该 Yarn-App 想怎么执行则是该 Yarn-App 自己的事，YARN 不需要（也不想）知道它的逻辑。比如图 5-10 中的 YARN 集群中正在运行一个 Yarn-App，该 Yarn-App 调用了额外四个 Container 来执行了本 Yarn-App。理解时可将 YARN 想象成 Win7，Yarn-App 比做 QQ，显然，Win7 上不但可以运行 QQ，还可运行 notepad，且 Win7 自身也不知道 QQ 内部到底是什么逻辑。

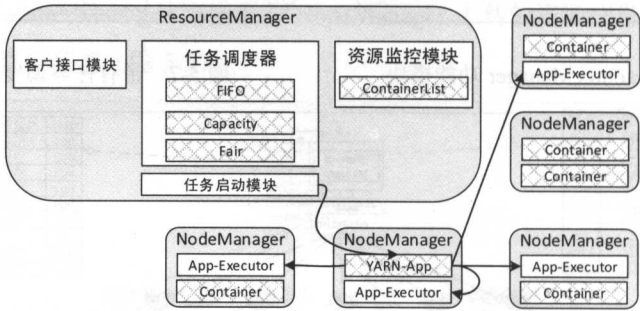


图 5-10 RM 在各 slave 上发起 Yarn-App

至此，分布式资源管理器引例结束，通过上述步骤构建的 preYARN，具备了分布式操作系统最基本的功能。当然，现实中 YARN 的实现机制则更加复杂，比如它的资源本地化机制、机架感知、Container 再分配策略、容量调度策略、Security 安全机制等，都有其精妙之处，但其架构的基本思路和本节很类似，读者可通过本节理解分布式资源管理架构思想，下面章节将深入 YARN 架构。

5.2 YARN 简介

通过上述内容，我们了解了 YARN^[3]应具有的基本功能模块，但是，作为大数据平台基础性组件，仅知道这些基本内容还不够，还要深入这些模块，本节主要讲述 YARN 的体系架构，从内部剖析 YARN。

5.2.1 基础概念

下面是 YARN 中涉及的主要概念，各概念在图 5-11 中都有所体现，阅读时，有些概念可能会难以理解，这是因为这些概念（包括图 5-11）包括了本章所有知识点，读者只需要对这些概念有所了解，在后续章节讲述 YARN 体系架构时，还会对这些概念进行深

2. NodeManager

YARN 从进程, 集群中每个从节点上都要部署一个 NodeManager, 主要负责管理本机资源。在执行 Yarn-App 时, NodeManager 还要启动 Container。

Container 是各类不同资源集, 它是一个逻辑概念, 比如, 可称单机上的<1 核 CPU、2G>为一个 Container。slave 上的 NodeManager 以 Container 形式向 ResourceManager 汇报本机资源信息, ResourceManager 以 Container 形式汇总集群内所有资源信息。当 Client 向 RM 提交 Yarn-App 时, Yarn-App 以 Container 为单位向 RM 申请资源, RM 则以 Container 形式向 Yarn-App 分配资源。

3. Client

YARN 服务使用者, 可以是 Web 用户、命令行用户或者程序用户, YARN 可向 Client 提供各类服务。

4. Yarn-App

Yarn-App 为在 YARN 上运行的应用程序, YARN 上可以运行各式各样的应用程序, 比如可以在 YARN 上运行 MR-App、Spark-App、Hive-App、Giraph-App 等。理解时, 可将 YARN 比做 Win7, MR-App、Spark-App、Hive-App、Giraph-App 比作 QQ、notepad、Word、Photoshop, 显然, Win7 上可以同时运行各类应用。同理只要 Yarn-App (MR-App、Spark-App、Hive-App、Giraph-App) 实现 YARN 协议, 至于该 Yarn-App 内部到底是什么逻辑, 是该 Yarn-App 自己的事, YARN 并不关心。

(1) ApplicationMaster

Client 提交的 Yarn-App 控制单元, 为方便理解, 可将 YARN 比做 Win7, Yarn-App-0 比作 QQ, Yarn-App-1 比作 notepad。显然 QQ 的逻辑功能和 notepad 完全不同, 反应到 YARN 中就是 Yarn-App-0 (QQ) 有其对应的 ApplicationMaster, Yarn-App-1 (Notepad) 也有其对应的 ApplicationMaster, 两个 ApplicationMaster 之间无任何关系。图 5-11 所示的 YARN 集群上只运行一个 Yarn-App, 故只有一个 ApplicationMaster。

(2) AppExecutor

Yarn-App 的实际执行者, 是 Container 的实体化 (在 Container 上执行代码即成为 App-Executor), 读者可能会有疑惑, 既然 ApplicationMaster 已经在执行 Yarn-App 了, 那么为何还需要此模块。道理很简单, 这是因为我们的终极目标是并行化, 那么如何才能实现并行化呢, 就是通过 ApplicationMaster 持有一系列 App-Executor 进行的。比如图 5-11 中, ApplicationMaster 会指挥其下属的四个 App-Executor 并行执行任务。

读者可能还会有疑惑，为何不再省去一步，让 RM 直接指挥各 App-Executor。这是因为集群很大，任务很多，RM 会成为瓶颈。比如某组织一个项目经理（Project Manager，PM）和一万名员工（employee）。当仅有一个 Project 时，PM 指挥所有 employee 有条不紊进行；现有两个 Project 要同时开工，则 PM 指挥部分 employee 做第一个任务，同时指挥部分 employee 做第二个任务。当一千个任务要同时开工，PM 要同时指挥一千个任务开工，PM 自身崩溃。诚然，此时可以聘用多个 PM，公司负责接任务，PM 负责指挥员工做任务。反应到分布式中就是 YARN 情形，YARN（公司）负责接任务，各 ApplicationMaster（PM）负责指挥 employee 做任务。

5.2.2 物理拓扑

YARN 典型物理拓扑是在主节点上部署主服务 ResourceManager，从节点上部署从服务 NodeManager。不过，和 NameNode 一样，也可以为 ResourceManager 服务配置其热备节点，由于 RM 的“双机热备份”较为复杂，这里不再讲述。

图 5-12 为 YARN 典型物理拓扑，从图中可以看出，YARN 采用 master/slave 架构，主节点 cmaster0 机运行主服务 ResourceManager，从节点 cslave0~N 运行从服务 NodeManager。使用 YARN 服务的用户都是 YARN 的客户端，比如图中的 iclient0、iclientN 即为 YARN 客户端（图 5-12）。

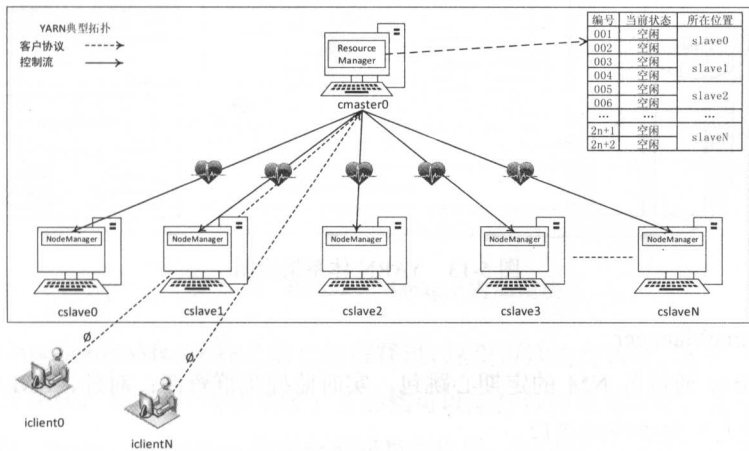


图 5-12 YARN 典型物理拓扑图

从节点上的 NodeManager 会不停监控本机资源状态并定期向 ResourceManager 上报这些状态信息（心跳包），在这些定期心跳包里 NodeManager 会写明本机 Container 个数及其当前使用情况。ResourceManager 则通过汇合所有 NodeManager 信息形成一张（以

Container 为单位) 集群全局资源表。

5.2.3 体系架构

实际上, YARN 只包含资源管理层, 程序执行层属于 Yarn-App (MapReduce、Spark、Hive 等), 而这些内容将会在后续章节中逐一讲解。不过了解 Yarn-App 执行流程有助于理解 YARN, 故本节第二部分, 编者列举了几个 Yarn-App 示例。

1. 集群资源管理层

YARN 采用 master/slave 架构, 主节点上运行主服务 ResourceManager, 从节点上运行从服务 NodeManager (图 5-13)。Client 为安装 YARN 客户端协议的实体机器, 集群正常运行时, 只存在 RM 和 NM 两类实体进程, Client 并不运行。

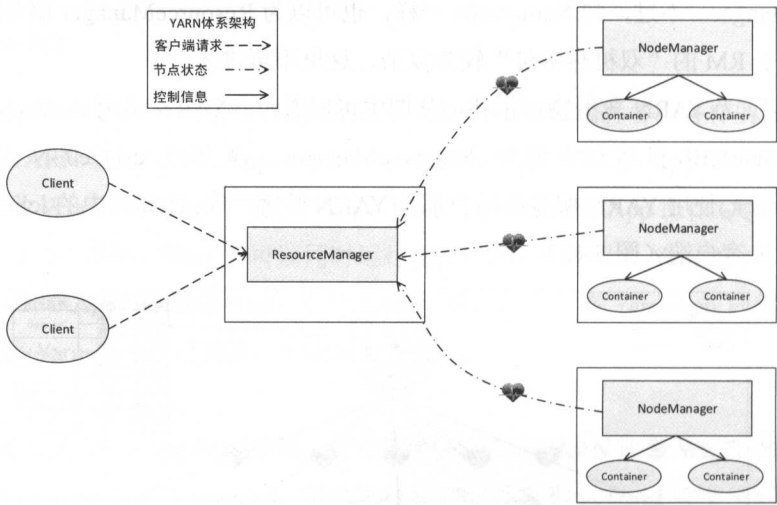


图 5-13 YARN 体系架构图

- ResourceManager

对内, RM 通过各 NM 的定期心跳包, 实时监控集群资源; 对外, RM 须为客户端 (程序或自然人) 提供访问接口。

- NodeManager

对内, NM 以 Container 为单位管理本机资源, 对外, NM 须定期向 RM 汇报心跳包, 当然, NM 也会根据 RM 的回复心跳包向 Container 发出相应指令。

YARN 中的 Container 为一逻辑概念, 可看成各类资源的一个逻辑集合, 比如可称<1核、1G>(<CPU 内核数、内存量>)为 1 个 Container, 此时双核 2G 机器包含两个 Container;

再比如也可称<2 核、2G>为 1 各 Container，则此时 2 核 2G 机器仅可分为一个 Container。在整个集群内，只要标准统一，那么各机 Container 数固定。引入 Container 概念后，可大大方便 NM 管理本机资源，RM 汇总集群资源。

• Client

非活动实体，可以是命令端、Web 端或程序端，制造工具的目的都是使用工具，Client 就是 YARN 的使用者。

由上述简介可以看出 RM 和 NM 是 YARN 核心，下面分别讲述这两个部分。

1) ResourceManager

RM 内部包含诸多模块（图 5-14），按功能，可将这些模块分为客户端模块，资源监控模块，调度模块和任务启动模块。不过，RM 这些模块都是为与 Client、AppMstr 和 NM 交互。故要讲述 RM，就要讲述 RM 与 Client、AppMstr、NM 之间相互关系，图 5-15 即为这三者（Client、AppMstr、NM）和 RM 之间相互关系图。

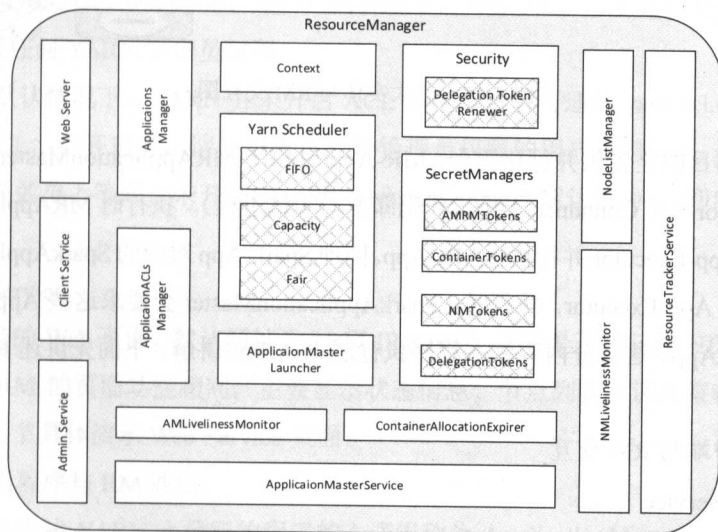


图 5-14 ResourceManager 内部模块

图中的 ApplicationMaster 在之前已经解释过，这里再次进行讲述，既然可以将 YARN 看成一个操作系统，那 OS（操作系统）上必然可以运行各种应用程序。就像是 Win7 上可以运行 QQ、Word、Photoshop 等完全不同类型程序，YARN 上也可以运行各类完全不同的应用程序，比如符合 MapReduce 范式的 MR-App，再比如 Spark 类文件挖掘程序 Spark-App。就像是 Win7 下 Word 有 Word 的逻辑，Photoshop 有 Photoshop 的逻辑一样，YARN 下的 MR-App 有 MR-App 的逻辑，Spark-App 有 Spark 的逻辑，即程序逻辑部分由程序自己定义，反应在代码上就是 ApplicationMaster 代码集，代码执行时显示的进程名

也为 ApplicationMaster。

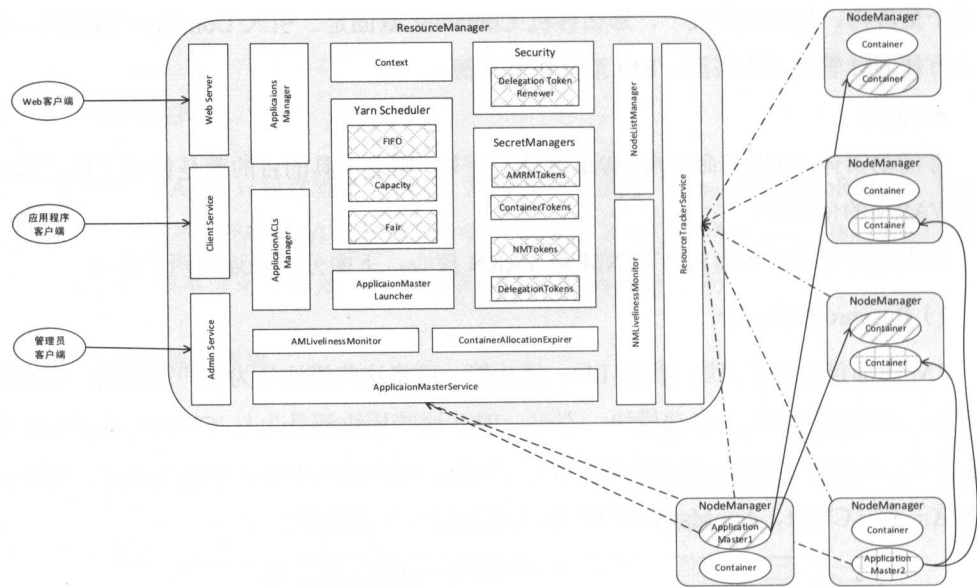


图 5-15 三大实体与 RM 交互图

YARN 的目的是提供并行化平台，MR-App 对应的 MRApplicationMaster 会持有一系列 App-Executor（在 Container 上执行代码即变为实体进程），执行时 MRApplicationMaster 会要求这些 App-Executor 并行执行 MR-App。同理 Spark-App 对应的 SparkApplicationMaster 也持有一系列 App-Executor，执行时，SparkApplicationMaster 会要求这些 App-Executor 并行执行 Spark-App。这部分内容将在程序执行层再次细致讲解，下面先讲述 RM 包含的各大组件。

(1) 客户端与 RM 交互

● Client Service

模块 Client Service 主要面向应用程序，在其内部，Client Service 实现了客户端到 RM 的接口 ApplicationClientProtocol。Client Service 会处理所有来自客户端到 RM 的 RPC 操作。通过此模块，应用程序（一种特殊的客户端）能够完成：

- 提交应用程序
- 终止应用程序
- 获取应用程序当前状态信息

此外 Client Service 也是验证集群安全协议的地方，当 YARN 开启 Kerberos 或授权 Token 认证时，Client Service 会验证 Client 提交过来的应用程序是否满足 Kerberos 或授权 Token。从而屏蔽掉了非授权用户提交的应用程序，为 RM 核心组件提供了一层安全墙。

● Administration Service

和 Client Service 面向应用程序不同, Administration Service 主要面向 YARN 管理员。之所以将此模块独立出来,是因为给 YARN 管理命令提供高优先级,让 RM 优先执行管理员命令,从而防止 YARN 管理员级别命令被一般用户级应用程序饿死,Administration Service 主要功能是:

- 刷新 YARN Scheduler 模块的调度策略与队列
- 刷新 NodesListManager 模块维护的 NM 节点列表
- 管理用户组,管理 ACL (Access Control List)

和 Client Service 类似, Administration Service 模块也会过滤非法请求,不过 Administration Service 优先使用 ACL 进行认证,比如编者以 Allen 用户在 iclient0 上向 littleCstor 提交了一个编号为 application_1436523327981_0006 的 YARN 任务。非法用户 Jack 看到后从随即从其所在机器 xclient7 机上通过 YANR 命令终止编者的 application_1436523327981_0006。为防止这种现象,可在 ACL 中配置拒绝操作,直接剥夺 xclient7 机任何 YARN 管理员权限。

不过,默认情况下, YARN 并未开启 ACL 认证,读者可通过 yarn.acl.enable 属性为集群开启 ACL,在开启后,请务必将可授予管理员权限的用户和组写入 yarn.admin.acl 标签,否则在效果上等同于未开 ACL (因为 yarn.admin.acl 默认值为*,即所有用户皆为管理员)。

● Web Service

RM 自带的 Web 页面,默认网址为“RM-IP:8080”,读者是否还记得 HDFS 默认 Web 页面,此处 RM 的页面功能相同,主要显示状态信息、节点列表和调度策略,编者将在 YARN 接口章节具体演示 Web Service 页面。

(2) 应用程序与 RM 通信

前面已经讲述 YARN 上运行的程序的主逻辑称为 ApplicationMaster,一旦一个应用程序成功通过 Client Service, ApplicationMasterLauncher 会选择一空闲 Container,并在此 Container 上执行此应用程序主服务,此时称执行此应用程序主逻辑的进程为 ApplicationMaster (如图中的 ApplicationMaster1),下面介绍 ApplicationMaster 与 RM 之间的通信模块。

● ApplicationMasterService

该模块实现了 ApplicationMasterProtocol 协议,是 ApplicationMaster 和 RM 通信的唯一协议,主要用来响应所有 ApplicationMaster 的请求,如图中该模块负责响应 ApplicationMaster1 和 ApplicationMaster2 的请求。ApplicationMaster 和 RM 通过 ApplicationMasterService 交换的信息主要分为下述三类:

- 响应 ApplicationMaster 的注册、取消、终止等操作
- 验证来自不同 ApplicationMaster 的各类请求
- 获取来自所有运行 ApplicationMaster 的 Container 分配和释放请求, 并异步转发给 YARN 调度器

● AMLivelinessMonitor

该模块负责跟踪每个 ApplicationMaster, 如果某 ApplicationMaster 超过十分钟(可配置)还未上传心跳包, AMLivelinessMonitor 认为此 ApplicationMaster 超时失败, 此 ApplicationMaster 申请的诸多 Container 也会被标记死亡并等待 RM 回收。不过 RM 会重新调用这个应用程序, 并在一新的空闲 Container 上运行此应用程序对应的 ApplicationMaster 实例, 默认情况下, 最多允许两次这样的尝试。

(3) NodeManager 与 ResourceManager 通信

● ResourceTrackerService

该模块实现了 ResourceTracker 接口, 主要负责响应所有 NodeManager 定期上传的心跳 RPC, 比如:

- 新 NodeManager 注册
- 接收已注册 NodeManager 的心跳 RPC
- 安全认证(确定 NodeManager 是否合法)
- 转发合法 NodeManager 心跳包(内含 Container 列表)至调度器

对于那些由非法或退役 NodeManager 发来的心跳包, ResourceTrackerService 会拒绝它们的注册请求; 对于合法 NodeManager 发来的心跳包, ResourceTrackerService 会将 Container 信息转发至 Yarn Scheduler 模块, Yarn Scheduler 会根据节点空闲状态以及应用程序请求做出调度。

● NMLivelinessMonitor

该模块会跟踪每个 NodeManager, 如果某 NodeManager 超过十分钟(可配置)还未上传心跳包, 该 NodeManager 被设定为超时死亡, 其上的正在运行的一个或多个 Container 也被标记为死亡, 这些 Container 会被其对应 ApplicationMaster 重新分配至其他 NodeManager 的 Container。

对于被标记为死亡的 NodeManager, 一旦该 NodeManager 重启后将会重新加入集群并参与 Container 调度。

● NodesListManager

该模块是各 NodeManager 在 RM 内存中的一个集合, RM 通过该集合来管理 NM。

上述内容主要讲解 RM 和其他组件的交互情况, 下面讲述 RM 核心组件。

- ApplicationManager

该模块主要管理已提交的应用程序，比如检查应用程序提出的资源请求是否合法，记录已完成应用程序基本信息（至于已完成应用程序的完整执行输出由 HistoryServer 托管）。

- ApplicationMasterLauncher

客户端应用程序穿过 Client Service 后，ApplicationMasterLauncher 会向 Yarn Scheduler 模块注册应用程序并申请一个空闲 Container，ApplicationMasterLauncher 随后会在此 Container 上发起此应用程序的 ApplicationMaster。读者应当注意，此应用程序随后的 Container 由 ApplicationMaster 自行申请并拉起。

- YarnScheduler

在本章开始时，编者已经讲述，操作系统核心功能之一是进程调度（也称 CPU 调度），同样 YarnScheduler 也是分布式操作系统 YARN 的最核心的模块。它主要负责给所有正在运行的应用程序分配资源，当 ApplicationMaster 感知到某应用程序结束时，YarnScheduler 会回收此应用程序占用的 Container。

当前 YARN 支持的调度策略为 FIFO、Fair 和 Capacity，其中 CapacityScheduler 为默认调度策略，通过配置 CapacityScheduler，可以实现几个组织共用一个大集群，不过这三大调度策略可以嵌套配置（通过队列），此外 YARN Scheduler 支持热插拔，可动态配置调度器。

RM 的另一大构成是安全性组件，主要包含如下几个模块。

- ContainerTokenSecretManager

当在 YARN 上运行应用程序时，此应用程序的 ApplicationMaster 需要权限才能调用 NodeManager 上的 Container。这时就需要 RM 的 ContainerTokenSecretManager 模块给 ApplicationMaster 提供一个系统类令牌集，ApplicationMaster 持有这些令牌集并依据这些令牌集赋予的权限，向某 NodeManager 提出要求，执行此 NodeManager 维护的 Container。通过授权，只有有权限的 ApplicationMaster 才能执行 Container。

注意 ApplicationMaster 申请 Container 依旧是到 YarnScheduler 申请的，只不过当需要具体使用某 NodeManager 上的 Container 时，它才需要这个权限。

- AMRTokenSecretManager

正如上文所示，ApplicationMaster 需要到 YarnScheduler 申请 Container，为了避免潜在的恶意程序模仿一个真正的 ApplicationMaster 向 YarnScheduler 申请 Container，RM 的 AMRTokenSecretManager 模块会为每个合法的 ApplicationMaster 生成一系列令牌，这样，只有合法的 ApplicationMaster 才能申请 Container。

- NMTOKENSecretManager

在应用程序执行过程中，ApplicationMaster 还需要和 NodeManager 进行通信，此时 ApplicationMaster 就需要使用 NMTOKEN 来管理和 NodeManager 的连接。

- DelegationTokenSecretManager

只有授权的客户端才能向 RM 提交应用程序，DelegationTokenSecretManager 负责给客户端生成代理令牌，该令牌会传递至想要和 ResourceManager 通信，但又没有启用 Kerberos 认证的应用程序。

- DelegationTokenRenewer

该模块在应用程序运行期间更新应用程序的令牌，直到该令牌不能再更新。

2) NodeManager

图 5-16 为 NodeManager 内部组件图，和 RM 一样，NodeManager 内部也可划分为一系列组件，每个组件完成特定功能。

NodeManager 的核心功能是管理 Container，当 YARN 上执行应用程序时，NodeManager 负责接受来自 ApplicationMaster 的启动或停止 Container 请求，对 ApplicationMaster 持有的令牌进行鉴权，管理 Container 执行依赖库，监控 Container 执行过程。

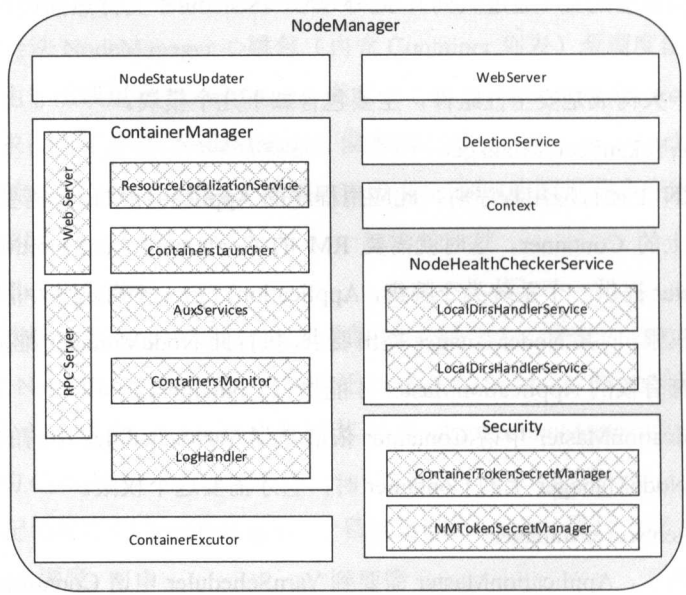


图 5-16 NodeManager 内部模块

ApplicationMaster 在启动、查看和关闭 Container 时都要和 NodeManager 通信，并借助 NodeManager 向 Container 发出相关查看或关闭指令。

NodeManager 主要包含如下几大功能模块。

- NodeStatusUpdater

在 NodeManager 刚启动时, NodeStatusUpdater 负责向 ResourceManager 注册并汇报本机资源、RPC 端口等信息。RM 则在返回心跳包中给出安全相关的 KEY, NodeManager 将用这个 KEY 为 ApplicationMaster 的 Container 请求做认证。

- ContainerManager

它是 NodeManager 的核心组件, 包含诸多子模块, 它们共同协作管理本机 Container。ContainerManager 的各个子模块功能非常关键, 比如 ResourceLocalizationService 负责将执行应用程序执行过程中资源下载到本地——资源本地化。资源本地化操作比较复杂, 读者只要知道, 当某单机执行整个 Yarn-App 的部分任务时, 该单机须下载所有执行相关资源。

- ContainerExecutor

该模块主要负责和底层操作系统交互, 来启动、关闭并清理 Container 相关进程。

- NodeHealthCheckerService

该模块通过定期运行健康检查脚本, 来检测节点当前健康状态, 当系统健康状态发生改变时, 该模块会将更改信息发送给 NodeStatusUpdate, 而 NodeStatusUpdate 则将更改信息传递至 ResourceManager。

2. 应用程序执行层

YARN 上的应用程序 Yarn-App 也采用 master/slave 架构, master 进程称为 ApplicationMaster, slave 进程称为 App-Executor。

我们知道可以在 YARN 上运行不同类型的应用程序, 如可以在 YARN 上运行 MR-App (MapReduce 应用程序)、Spark-App (Spark 应用程序)、Giraph-App (Giraph 应用程序)。显然, 这些应用程序主控模块 ApplicationMaster 逻辑完全不同, 故为进一步区分不同的 ApplicationMaster 和 App-Executor, 编者分别在 ApplicationMaster 和 App-Executor 前面加类型前置符, 比如 SparkAppMaster/Spark-Executor、MRAppMaster/MR-Executor、GiraphAppMaster/Giraph-Executor。

以 Spark-App 为例, 程序运行时, SparkAppMaster 会指挥其附属的 SparkExecutor 并行执行用户提交的 Spark-App。图 5-17 即为正在执行一个 Spark-App 的集群状态图, 该图只专注程序执行层, 并未显示任何其他进程, 方便读者理解。程序执行时, 执行进程由 SparkAppMaster0 和 SparkExecutor 构成, SparkAppMaster0 持有三个 SparkExecutor, 整个执行步骤皆由 SparkAppMaster0 指挥, 各个 SparkExecutor 并行执行 SparkAppMaster0 分配给本 Executor 的任务。总之 SparkAppMaster0 会统一指挥其附属的三个 SparkExecutor

来运行用户提交的 Spark-App。

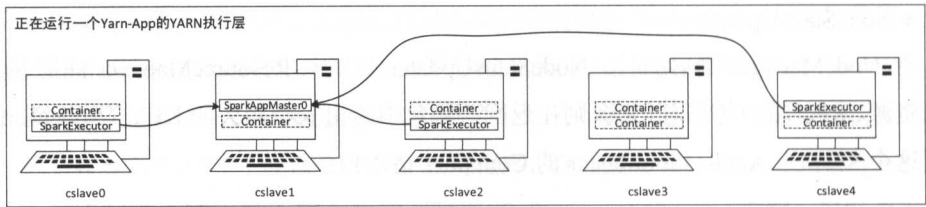


图 5-17 正在运行一个 Yarn-App 的程序执行层

同理，当 YARN 上执行 MR-App 时，其执行步骤也是 MRAppMaster1 指挥其附属的 MRExecutor 并行执行用户提交的 MR-App。图 5-18 显示了正在执行两个应用（Spark-App、MR-App）的程序执行层，其中 SparkAppMaster0 为 Spark-App 的主进程，该进程持有三个 SparkExecutor；MRAppMaster1 为 MR-App 的主进程，该进程持有两个 MRExecutor。SparkAppMaster0 和 MRAppMaster1 申请的 Container 数量由用户程序和 Scheduler 统一决定。

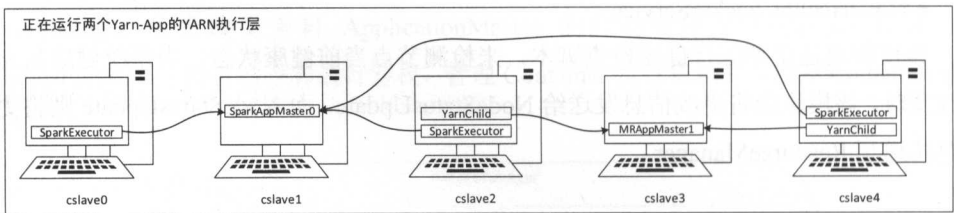


图 5-18 正在运行两个 Yarn-App 的程序执行层

3. 工作机制

YARN 集群正常运行时，只存在 RM 和 NM 进程。当客户端向 YARN 提交 Yarn-App 时，客户端会启动 Client 进程来向 RM 提交任务。RM 在收到客户提交的程序后，会启动该客户程序的 ApplicationMaster。ApplicationMaster 会负责该用户程序的一切执行事宜。详细步骤可描述如下，请读者参考图 5-19 仔细理解。

Step1 iclient0 向 ResourceManager 注册应用程序（图中步骤①）。

Step2 ResourceManager 查看集群资源配置表，选定一空闲 Container，比如选中 cslave1 上的 Container（图中步骤②）。

Step3 ResourceManager 指示 cslave1 上的 NodeManager 在 Container 里启动 SparkAppMaster0（图中步骤③）。

Step4 SparkAppMaster0 根据 SparkAppBusinessLogic（用户代码）向 ResourceManager 申请一定数量的 Container（图中步骤④）。

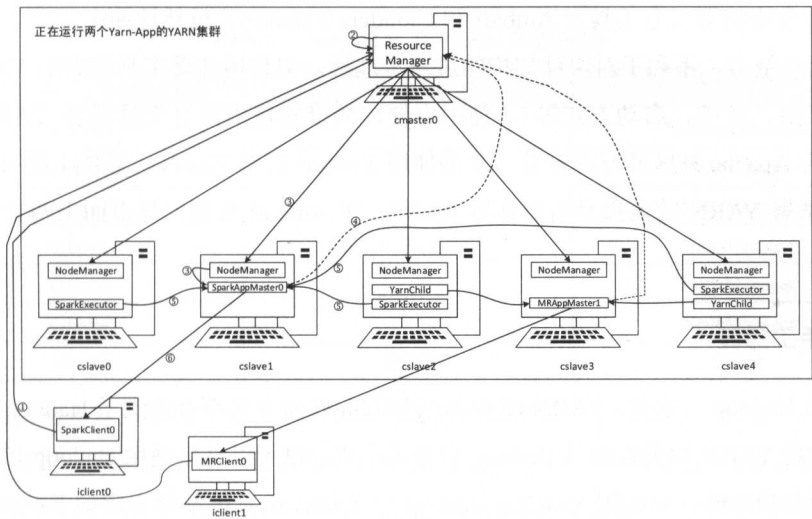


图 5-19 正在运行两个 Yarn-App 集群状态图

Step5 SparkAppMaster0 指示 NodeManager 在这些 Container 上启动 SparkExecutor(图中步骤④)。

Step6 隶属于本 SparkAppMaster0 的 SparkExecutor 向本 SparkAppMaster 注册(图中步骤⑤)。

Step7 SparkAppMaster0 根据用户编写的 SparkAppBusinessLogic 指挥 SparkExecutor 并行执行用户任务(图中步骤⑤)。

Step8 任务执行过程中,各 SparkExecutor 向 SparkAppMaster0 汇报任务执行进度(图中步骤⑤)。

Step9 任务执行过程中, SparkAppMaster0 向 SparkClient0 汇报任务执行进度(图中步骤⑥)。

显然图中的 YARN 集群正在运行两个 Yarn-App, 请读者参照 Spark-App 执行过程, 分析 MR-App 执行步骤。

5.2.4 集群部署

YARN 的部署方式主要为手工部署和工具部署, 手工部署最大优势是加深用户对 YARN 体系架构的认知, 不过手工部署的 YARN 在用户权限分配、环境变量设置、服务启动设置、命令调用方式等方面存在巨大缺点, 不但增加了维护成本, 还可能直接导致上层程序因无充足权限而无法使用 YARN。

目前主流的第三方工具为 Ambari 和 Cloudera Manager, 虽然这两大部署工具会屏蔽大部分组件细节, 不利于对组件架构的进一步理解, 但使用部署工具部署的 YARN 配置标准 (权限、环境、启动方式等), 使用方便, 故推荐使用部署工具部署 YARN。由于 Ambari 为 Apache 社区开发并推荐, 本书使用 Ambari 部署 YARN。需要注意的是, 使用 Ambari 部署 YARN 的前提是需要部署 HDFS, 在 Ambari 部署向导页面上两者同时选中即可。

1. 手工部署

作为 Hadoop 一部分, YARN 相关 Jar 包和 Shell 命令集都合并到了 Hadoop 里, 我们可以手动将 YARN 相关资源从 Hadoop 中分离出来, 也可以直接使用 Hadoop 包, 不过在进行配置和启动时, 只配置 YARN, Hadoop 中 YARN 的具体部署步骤如下:

Step1 制定部署规划。

Step2 准备硬件机器和 OS 环境。

Step3 配置单机 OS 环境和集群环境。

Step4 解压 Hadoop, 配置 HDFS、YARN。

Step5 初始化 HDFS、开启 HDFS。

Step6 开启 YARN、测试 YARN。

2. Ambari 部署

使用 Ambari 部署 YARN 可以说是一键操作, 难点几乎都在 Ambari 工具本身部署上, 以下步骤从无到有, 简单介绍了 Ambari 自身部署和使用 Ambari 部署 YARN 的大概步骤:

Step1 制定部署规划。

Step2 准备硬件机器和 OS 环境。

Step3 配置单机 OS 环境和集群环境。

Step4 部署 ambari-server。

Step5 使用 ambari-server 部署 HDFS、YARN。

例 1 请使用 Ambari 为 littleCstor 部署 YARN。

解 由于大数据平台涉及太多组件, 故部署之前最好制定一个完备的部署计划, 否则极有可能导致各机角色分配过乱, 甚至是部署失败。

表 5-1 为 littleCstor 上 YARN 的部署规划, YARN 本身只有主、从、客户端三个角色, 请读者在部署 YARN 之前, 也像编者这样, 制定部署规划。

表 5-1 littleCstor 上 YARN 部署规划

机 器	角 色	服 务
cmaster1	主服务	ResourceManager
cslave0	从服务	NodeManager
cslave1		NodeManager
cslave2		NodeManager
cslave3		NodeManager
cslave4		NodeManager
iclient0	客户端	Client

本题以第 3 章为前提，只给出 YARN 部署规划表（表 5-1）和部署效果图（图 5-20），具体部署过程请参见第 3 章。读者须注意，图 5-20 中 iclient0 到 cmaster1 或 cslave0、2 的链接（图中 Ø 表示）实际上并不存在，也就是 iclient0 并不需要向任何机器汇报心跳包，只有当 iclient0 需要使用 YARN 服务时，它才会主动连接 cmaster0 或 cslaveX。

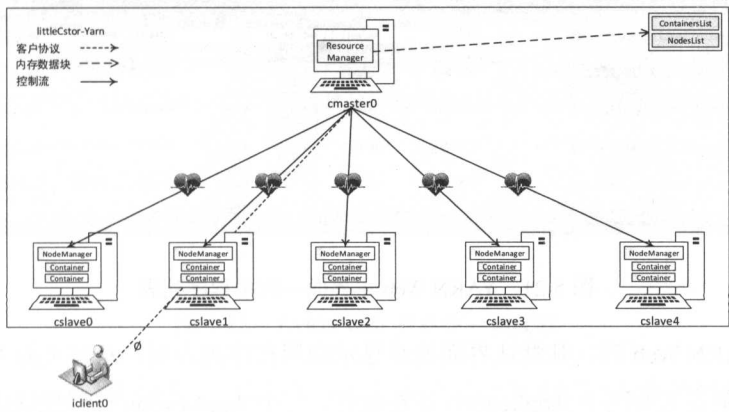


图 5-20 littleCstor—YARN 效果图

5.3 YARN 接口

作为大数据处理领域最常用的资源管理器，针对不同的上层应用，YARN 提供了三类统一访问接口^[4]，分别为：

- YARN 自带 Web 接口
- YARN Shell 接口
- YARN Java API 接口

Web 接口主要为管理员提供，从页面上，管理员能看到“集群统计信息”、“调度器”、

“应用程序列表”等功能模块，此页面只支持读，不支持写操作。

Shell 接口主要针对 YARN 管理员，通过 Shell 接口，管理员能够查看 YARN 系统级别统计信息，提交 Yarn-App 等，5.4 节重点介绍 YARN 的 Shell 接口。

YARN Java API 面向 Java 开发工程师，程序员可以通过该接口编写 Yarn-App，5.5、5.6 节将重点讲述 YARN 编程。

由于 YARN Web 接口内容较少，此处直接讲解。YARN Web 的默认地址是“RMIP:8080”，由于 littleCstor 中 RM 机器名为“cmaster0.cloudlab.njupt.edu”，故此处在浏览器中地址栏输入“cmaster0.cloudlab.njupt.edu:8080”即可进入 YARN Web 主界面(图 5-21)。

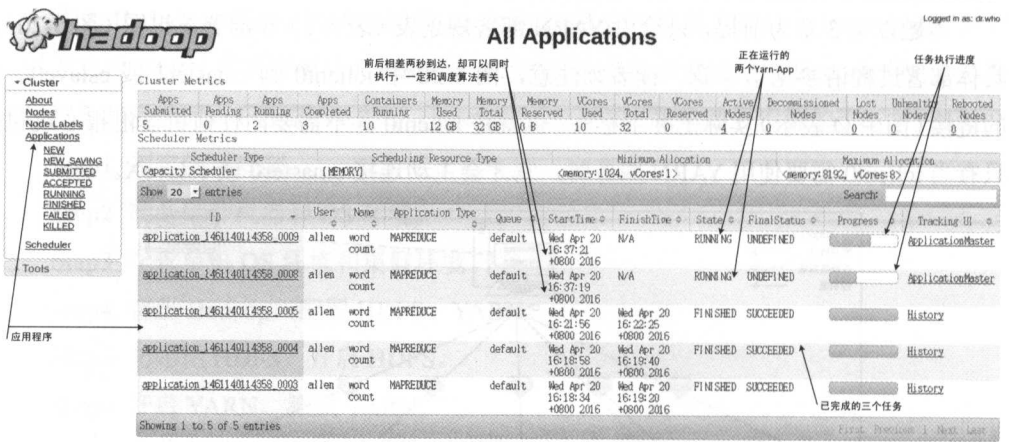


图 5-21 YARN Web 主界面—应用程序列表

进入 YARN Web 后，其默认界面就是显示应用程序列表页，从该页面上可以看出，此时 YARN 集群上有两个 Application 正在运行，三个 Application 已经完成，显然，正在运行的两个（application_1461140114358_0008 和 application_1461140114358_0009）任务之间相互不干扰，同时也在执行。也就是，application_1461140114358_0008 程序内部在并行执行着，外部各个 application 之间也是并行（机器足够多）执行的，在这点上，YARN 很像多用户多任务分时操作系统。

点击 Scheduler 选项，页面会导航至调度器页面（图 5-22），从图中可以出，此时集群采用 Capacity 调度策略，该策略（中文翻译为承载力调度策略）思想是：不管集群当前有多少个 application，只要集群内还有计算资源，都分配给这些应用程序，以确保这些应用程序同时运行。

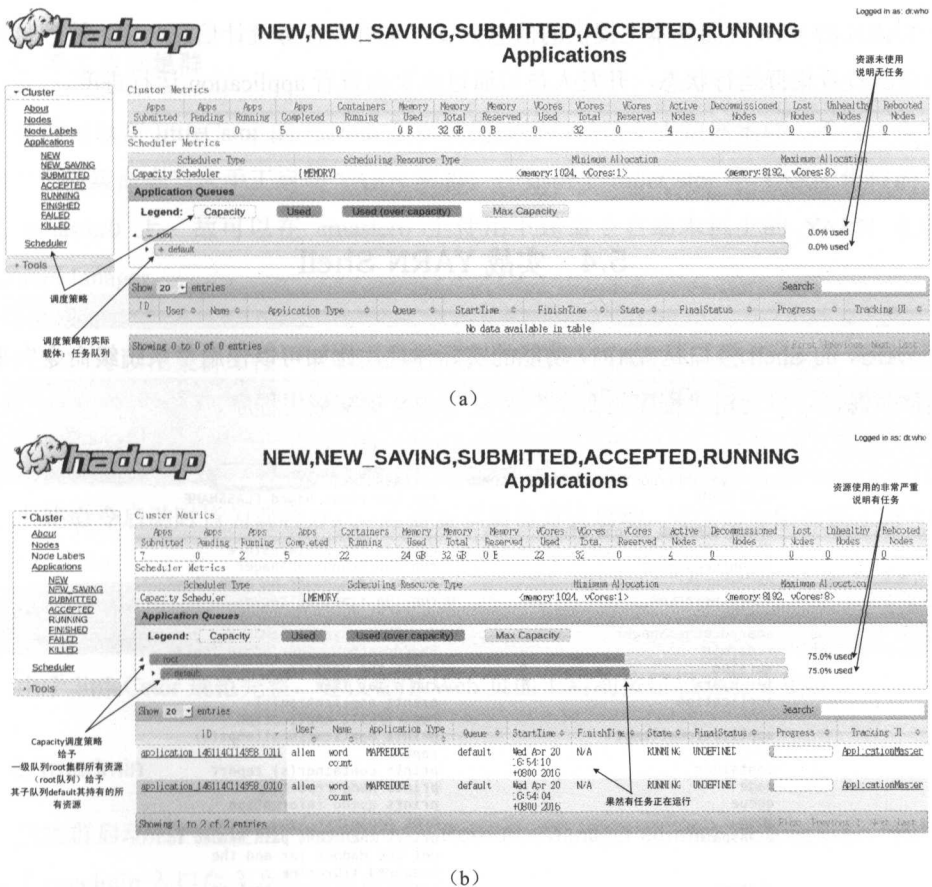


图 5-22 YARN Web 主界面—调度器

除了 Capacity 调度策略外，目前 YARN 还支持 Fair 和 FIFS（先来先服务），特别地，可以将这几种调度策略嵌套使用，以实现复杂的多级队列调度策略。

从 YARN Web 页面上还可看到 YARN 集群“系统级统计信息”（图 5-23），比如当前集群活跃节点数，YARN 版本信息等。

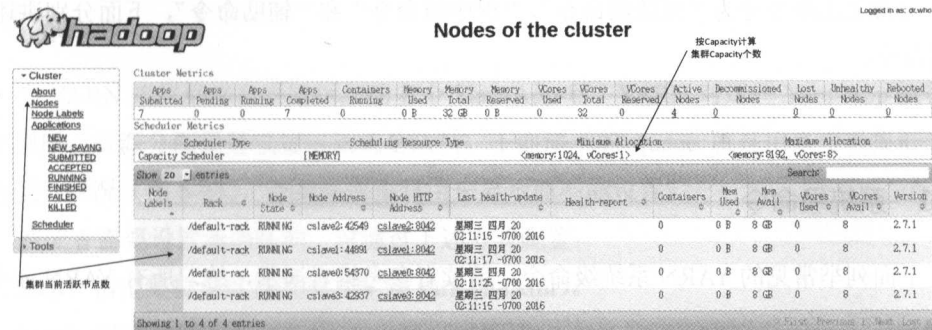


图 5-23 littleCstor—YARN 效果图

YARN 的 Web 页面基本上提供的信息主要以集群和任务统计信息为主，管理员可通过该页面查看集群运行状态，开发人员可通过该页面查看 application 运行进度。

5.4 实战 YARN Shell

YARN 的 Shell 接口是 YARN 功能的实际体现，比如可以使用“系统级命令”来管理集群资源，可以使用“程序级命令”来向 YARN 提交应用程序。

```
[allen@cmaster0 hadoop-2.7.1]$ bin/yarn
Usage: yarn [--config confdir] [COMMAND | CLASSNAME]
CLASSNAME                                run the class named CLASSNAME
or
where COMMAND is one of:
resourcemanager -format-state-store    deletes the RMStateStore
resourcemanager                               run the ResourceManager
nodemanager                               run a nodemanager on each slave
timelineserver                             run the timeline server
rmadmin                                   admin tools
sharedcachemanager                         run the SharedCacheManager daemon
scmadmin                                  SharedCacheManager admin tools
version                                  print the version
jar <jar>                                run a jar file
application                               prints application(s)
                                         report/kill application
applicationattempt                         prints applicationattempt(s)
                                         report
container                                prints container(s) report
node                                     prints node report(s)
queue                                   prints queue information
logs                                  dump container logs
classpath                             prints the class path needed to
                                         get the Hadoop jar and the
                                         required libraries
cluster                                prints cluster information
daemonlog                             get/set the log level for each
                                         daemon

Most commands print help when invoked w/o parameters.
[allen@cmaster0 hadoop-2.7.1]$
```

图 5-24 YARN 命令统一入口

YARN Shell 入口统一，其所有命令的第一标签都在“yarn”命令下，图 5-24 为 yarn 命令行统一入口。

可将上述命令分为“系统级命令”、“程序级命令”和“辅助命令”，下面分别讲述这三类命令。

5.4.1 系统级命令

下面列举常见的 YARN 系统级命令，请读者参考编者演示，练习所有 YARN 命令。

1. 手工启动集群

虽然可以在 littleCstor 的主 Web 页面使用按钮启动整个大数据集群，不过，也可以使用“系统级命令”来手工启动 YARN 集群，比如由于 littleCstor 中 YARN 组件的主服务在 cmaster0 上，故可以在 cmaster0 上使用下述命令启动本机上的 YARN 主服务 ResourceManager。

```
[yarn@cmaster0 ~]$ yarn resourcemanager #cmaster0 机,yarn 用户,启动 ResourceManager
```

同理，可以在集群中任一 slave 机上启动 YARN 从服务 NodeManager，举例如下：

```
[yarn@cslave0 ~]$ yarn nodemanager #cslave0 机,yarn 用户,启动 NodeManager
```

```
[yarn@cslave1 ~]$ yarn nodemanager #cslave1 机,yarn 用户,启动 NodeManager
```

```
[yarn@cslave2 ~]$ yarn nodemanager #cslave2 机,yarn 用户,启动 NodeManager
```

上述命令启动时皆为前台方式，若需要以后台方式启动，可在命令后加提示符“&”，如：

```
[yarn@cmaster0 ~]$ yarn resourcemanager &
```

```
[yarn@cslave0 ~]$ yarn nodemanager &
```

由于 littleCstor 规范完整，建议使用 Web 页面上启动按钮启动整个集群。

2. rmdadmin

当集群启动后，可使用“rmdadmin”（ResourceManager admin）来管理集群，图 5-25 演示了 rmdadmin 入口命令及其示例命令。

3. node

该命令主要用来查看集群当前节点信息，图 5-26 演示了该命令入口及其示例，当节点出现故障时，可以使用类似“yarn node -status cslave1:45648”命令单独查看某节点，当然寻找问题的唯一依据还是 log 日志。

4. queue

当 YARN 接收到来自不同用户的多个 Yarn-App 后，YARN 对这些 Yarn-App 执行的优先顺序、资源分配量等设置依据就是队列。下面演示的 queue 就是用来查看集群当前队列运行状况（图 5-27），不过该命令只用来显示而不能设置。

若读者需要设置新的队列，可通过下述方式配置：

Step1 指定集群调度器类型（配置 yarn-site.xml）。

Step2 配置该调度器（选定 capacity 时配置 capacity-scheduler，选定 fair 时配置 fair-scheduler）。

```
[allen@cmaster0 hadoop-2.7.1]$ bin/yarn rmadmin
Usage: yarn rmadmin
  -refreshQueues
  -refreshNodes
  -refreshSuperUserGroupsConfiguration
  -refreshUserToGroupsMappings
  -refreshAdminAcls
  -refreshServiceAcl
  -getGroups [username]
  -addToClusterNodeLabels [label1,label2,label3] (label splitted by ",")
  -removeFromClusterNodeLabels [label1,label2,label3] (label splitted by ",")
  -replaceLabelsOnNode [node1[:port]=label1,label2 node2[:port]=label1,label2]
  -directlyAccessNodeLabelStore
  -help [cmd]
```

Generic options supported are

- conf <configuration file> specify an application configuration file
- D <property=value> use value for given property
- fs <local|namenode:port> specify a namenode
- jt <local|resourcemanager:port> specify a ResourceManager
- files <comma separated list of files> specify comma separated files to be copied to the map reduce cluster
- libjars <comma separated list of jars> specify comma separated jar files to include in the classpath.
- archives <comma separated list of archives> specify comma separated archives to be unarchived on the compute machines.

The general command line syntax is
bin/hadoop command [genericOptions] [commandOptions]

```
[allen@cmaster0 hadoop-2.7.1]$ bin/yarn rmadmin -refreshQueues
16/04/17 02:04:53 INFO client.RMPProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8033
16/04/17 02:04:53 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-iaa
[allen@cmaster0 hadoop-2.7.1]$ bin/yarn rmadmin -refreshNodes
16/04/17 02:05:11 INFO client.RMPProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8033
16/04/17 02:05:11 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java
where applicable
[allen@cmaster0 hadoop-2.7.1]$ bin/yarn rmadmin -refreshSuperUserGroupsConfiguration
16/04/17 02:05:41 INFO client.RMPProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8033
16/04/17 02:05:41 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java
where applicable
[allen@cmaster0 hadoop-2.7.1]$ bin/yarn rmadmin -refreshUserToGroupsMappings
16/04/17 02:06:25 INFO client.RMPProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8033
16/04/17 02:06:25 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java
where applicable
[allen@cmaster0 hadoop-2.7.1]$
```

图 5-25 rmadmin 命令入口实例

```
[allen@cmaster0 hadoop-2.7.1]$
[allen@cmaster0 hadoop-2.7.1]$ bin/yarn node
16/04/17 02:34:28 INFO client.RMPProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
16/04/17 02:34:28 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform.
ltin-java classes where applicable
Invalid Command Usage :
usage: node
  -all Works with -list to list all nodes.
  -help Displays help for all commands.
  -list List all running nodes. Supports optional use of
        -states to filter nodes based on node state, all -all
        to list all nodes.
  -states <States> Works with -list to filter nodes based on input
        comma-separated list of node states.
  -status <NodeId> Prints the status report of the node.
```

```
[allen@cmaster0 hadoop-2.7.1]$ bin/yarn node -list
16/04/17 02:34:36 INFO client.RMPProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
16/04/17 02:34:36 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform.
ltin-java classes where applicable
Total Nodes:4
```

Node-Id	Node-State	Node-Http-Address	Number-of-Running-Containers
cslave0:49741	RUNNING	cslave0:8042	0
cslave2:52283	RUNNING	cslave2:8042	0
cslave1:60870	RUNNING	cslave1:8042	0
cslave3:44025	RUNNING	cslave3:8042	0

```
[allen@cmaster0 hadoop-2.7.1]$ bin/yarn node -status cslave2:52283
16/04/17 02:34:49 INFO client.RMPProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
Node Report :
Node-Id : cslave2:52283
Rack : /default-rack
Node-State : RUNNING
Node-Http-Address : cslave2:8042
Last-Health-Update : Sun 17/Apr/16 02:33:55:400PDT
Health-Report :
Containers : 0
Memory-Used : 0MB
Memory-Capacity : 8192MB
CPU-Used : 0 vcores
CPU-Capacity : 8 vcores
Node-Labels :
```

图 5-26 node 命令入口及其实例

```
[allen@cmaster0 hadoop-2.7.1]$ bin/yarn queue
16/04/17 02:49:02 INFO client.RMProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
16/04/17 02:49:03 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform.
ltin-java classes where applicable
Invalid Command Usage :
usage: queue
    -help                               Displays help for all commands.
    -status <Queue Name>               List queue information about given queue.
[allen@cmaster0 hadoop-2.7.1]$ bin/yarn queue -status root
16/04/17 02:49:15 INFO client.RMProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
16/04/17 02:49:15 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform.
ltin-java classes where applicable
Queue Information :
Queue Name : root
    State : RUNNING
    Capacity : 100.0%
    Current Capacity : .0%
    Maximum Capacity : 100.0%
    Default Node Label expression :
    Accessible Node Labels : *
[allen@cmaster0 hadoop-2.7.1]$ bin/yarn queue -status default
16/04/17 02:49:21 INFO client.RMProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
Queue Information :
Queue Name : default
    State : RUNNING
    Capacity : 100.0%
    Current Capacity : .0%
    Maximum Capacity : 100.0%
    Default Node Label expression :
    Accessible Node Labels : *
```

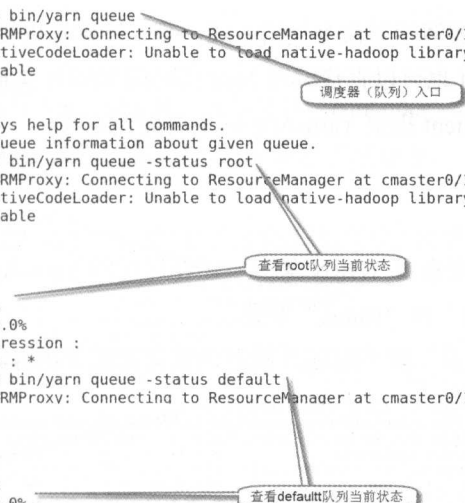


图 5-27 queue 命令入口及其实例

Step3 动态实时（不停止集群）刷新新配置的队列。

```
[yarn@iclient0 ~]$ yarn rmadmin -refreshQueues
```

Step4（可选） 实时查看队列资源使用量。

```
[yarn@iclient0 ~]$ yarn queue -status "队列名"
```

```
[yarn@iclient0 ~]$ yarn queue -status root
```

除了上述命令外，常见的系统级命令还有“cluster”、“sharedcachemanager”等，请读者自行练习。

5.4.2 程序级命令

既然 YARN 上可以运行 Yarn-App，那么其至少要提供提交 Yarn-App 的命令行接口，“程序级命令”即是 Client 向 YARN 提交 Yarn-App 的入口，此外 Client 还能使用该命令随时查看程序运行状态，主要的“程序级命令”罗列如下。

1. jar

Client 可以使用该命令向 YARN 集群提交 Yarn-App，比如下述命令就是用来向 YARN 提交 MapReduce 类型 Yarn-App PI，该代码实现了 BBP 算法的 MapReduce 并行化，可快速准确计算出 PI。

```
[allen@iclient0 ~]$ yarn jar jarFile [mainClass] args...
```

```
[allen@iclient0 ~]$ yarn jar hadoop-mapreduce-examples-2.7.1.jar pi 8 32
```

上述第一行主要演示该命令使用方式，第二行则是以提交 PI 为例，向 YARN 提交 PI 任务。虽然 Client 也可以直接编写 Java 代码向 YARN 集群提交 Yarn-App，不过，总的来说，jar 命令是 Client 提交 Yarn-App 的主要提交方式。

2. Application

该命令主要用来查看 YARN 集群中正在运行的 Yarn-App，图 5-28 演示了该命令及其下属“list”、“kill”和“status”参数。

图 5-28 中，“kill”和“status”执行的前提是集群中存在正在运行的 Yarn-App。

```
[allen@master0 hadoop-2.7.1]$ bin/yarn application
16/04/17 04:45:13 INFO client.RMPProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
16/04/17 04:45:13 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... us
ltin-java classes where applicable
Invalid Command Usage :
usage: application
      -appStates <States>

Works with -list to filter applications
based on input comma-separated list of
application states. The valid application
state can be one of the following:
ALL, NEW, NEW SAVING, SUBMITTED, ACCEPTED, RUN
NING, FINISHED, FAILED, KILLED
Works with -list to filter applications
based on input comma-separated list of
application types.
Displays help for all commands.
Kills the application.
List applications. Supports optional use
of -appTypes to filter applications based
on application type, and -appStates to
filter applications based on application
state.

- appTypes <Types>

-help
-kill <Application ID>
-list

-movetoqueue <Application ID>
Moves the application to a different
queue.
Works with the movetoqueue command to
specify which queue to move an
application to.
Prints the status of the application.

-status <Application ID>
[allen@master0 hadoop-2.7.1]$ bin/yarn application -list
16/04/17 04:46:11 INFO client.RMPProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
16/04/17 04:46:11 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... us
ltin-java classes where applicable
Total number of applications (application-types: [] and states: [SUBMITTED, ACCEPTED, RUNNING]):3
      Application-Id      Application-Name      Application-Type      User      Queue
      State      Final-State      Progress      Tracking-URL
application_1460883477129_0006      word count      MAPREDUCE      allen      default
      ACCEPTED      UNDEFINED      0%      N/A
application_1460883477129_0004      word count      MAPREDUCE      allen      default
      RUNNING      UNDEFINED      5%      http://cslave0:60999
application_1460883477129_0005      word count      MAPREDUCE      allen      default
      RUNNING      UNDEFINED      5%      http://cslave3:53238
[allen@master0 hadoop-2.7.1]$ bin/yarn application -kill application_1460883477129_0005
16/04/17 04:46:28 INFO client.RMPProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
16/04/17 04:46:28 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... us
ltin-java classes where applicable
Killing application application_1460883477129_0005
16/04/17 04:46:30 INFO impl.YarnClientImpl: Killed application application_1460883477129_0005
[allen@master0 hadoop-2.7.1]$ bin/yarn application -status application_1460883477129_0006
16/04/17 04:46:54 INFO client.RMPProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
16/04/17 04:46:54 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... us
ltin-java classes where applicable
Application Report :
  Application-Id : application_1460883477129_0006
  Application-Name : word count
  Application-Type : MAPREDUCE
  User : allen
  Queue : default
  Start-Time : 1460893561807
  Finish-Time : 0
  Progress : 50%
  State : RUNNING
  Final-State : UNDEFINED
  Tracking-URL : http://cslave3:38598
  RPC Port : 35991
  AM Host : cslave3
  Aggregate Resource Allocation : 67078 MB-seconds, 38 vcore-seconds
  Diagnostics :
[allen@master0 hadoop-2.7.1]$
```

application命令入口
作用：命令行为式管理Yarn App

显示：list
查看集群当前所存在运行的Yarn-App

显示：kill
强制停止某正在运行的Yarn-App

显示：status
查看正在运行的Yarn-App的运行进度

图 5-28 application 命令入口及其实例

3. container

当 YARN 集群上正在运行 Yarn-App 时，可以使用“container”命令查看本 Yarn-App 所有 Container 以及各 Container 执行状态。图 5-29 演示了该命令。

container命令统一入口

查看正在运行的Yarn-App
包含几个Container

演示: status
查看特定Container执行状态

```
[allen@master0 hadoop-2.7.1]$ bin/yarn container
16/04/17 07:28:50 INFO client.RMPProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
16/04/17 07:28:50 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Invalid Command Usage :
usage: container
  -help                Displays help for all commands.
  -list <Application Attempt ID>  List containers for application attempt.
  -status <Container ID>        Prints the status of the container.
[allen@master0 hadoop-2.7.1]$ bin/yarn container -list appattempt_1460883477129_0008_000001
16/04/17 07:32:20 INFO client.RMPProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
16/04/17 07:32:20 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Total number of containers :11
  Container-Id      Start Time      Finish Time      State      Host
Node Http Address
container_1460883477129_0008_01_000001  Sun Apr 17 07:31:57 -0700 2016      N/A      RUNNING      cs1a
ve3:44025      http://cslave3:8042
container_1460883477129_0008_01_000002  Sun Apr 17 07:32:12 -0700 2016      N/A      RUNNING      cs1a
ve3:44025      http://cslave3:8042
container_1460883477129_0008_01_000003  Sun Apr 17 07:32:12 -0700 2016      N/A      RUNNING      cs1a
ve3:44025      http://cslave3:8042
container_1460883477129_0008_01_000004  Sun Apr 17 07:32:12 -0700 2016      N/A      RUNNING      cs1a
ve3:44025      http://cslave3:8042
container_1460883477129_0008_01_000005  Sun Apr 17 07:32:12 -0700 2016      N/A      RUNNING      cs1a
ve3:44025      http://cslave3:8042
container_1460883477129_0008_01_000006  Sun Apr 17 07:32:12 -0700 2016      N/A      RUNNING      cs1a
ve3:44025      http://cslave3:8042
container_1460883477129_0008_01_000007  Sun Apr 17 07:32:12 -0700 2016      N/A      RUNNING      cs1a
ve3:44025      http://cslave3:8042
container_1460883477129_0008_01_000008  Sun Apr 17 07:32:13 -0700 2016      N/A      RUNNING      cs1a
ve2:52283      http://cslave2:8042
container_1460883477129_0008_01_000009  Sun Apr 17 07:32:13 -0700 2016      N/A      RUNNING      cs1a
ve2:52283      http://cslave2:8042
container_1460883477129_0008_01_000010  Sun Apr 17 07:32:13 -0700 2016      N/A      RUNNING      cs1a
ve2:52283      http://cslave2:8042
container_1460883477129_0008_01_000011  Sun Apr 17 07:32:13 -0700 2016      N/A      RUNNING      cs1a
ve0:49741      http://cslave0:8042
[allen@master0 hadoop-2.7.1]$ bin/yarn container -status container_1460883477129_0008_01_000006
16/04/17 07:32:37 INFO client.RMPProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
16/04/17 07:32:38 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Container Report :
  Container-Id : container_1460883477129_0008_01_000006
  Start-Time : 1460903532800
  Finish-Time : 0
  State : RUNNING
  LOG-URL : http://cslave3:8042/node/containerlogs/container_1460883477129_0008_01_000006/allen
  Host : cslave3:44025
  NodeHttpAddress : http://cslave3:8042
  Diagnostics : null
[allen@master0 hadoop-2.7.1]$
```

图 5-29 container 命令入口及其实例

5.4.3 其他辅助命令

YARN 上还有很多命令用来管理 YARN 集群和 Yarn-App，比如下述命令用来查看 YARN 版本：

```
[allen@iclient0 ~]$ yarn version
```

再比如显示 YARN 环境变量的“classpath”命令、显示某特定进程日志的“logs”命令等。在图 5-30 中，编者简单演示了这类命令，请读者自行练习所有 YARN 命令。

```
[allen@cmaster0 hadoop-2.7.1]$ bin/yarn version
Hadoop 2.7.1
Subversion https://git-wip-us.apache.org/repos/asf/hadoop.git -r 15ecc87ccf4a0228f35af08fc56de536e6ce657a
Compiled by jenkins on 2015-06-29T06:04Z
Compiled with protoc 2.5.0
From source with checksum fc0a1a23fc1868e4d5ee7fa2b28a58a
This command was run using /home/allen/hadoop-2.7.1/share/hadoop/common/hadoop-common-2.7.1.jar
[allen@cmaster0 hadoop-2.7.1]$ bin/yarn classpath
/home/allen/hadoop-2.7.1/etc/hadoop:/home/allen/hadoop-2.7.1/etc/hadoop:/home/allen/hadoop-2.7.1/etc/hadoop:/home/allen/hadoop-2.7.1/share/hadoop/common/lib/*:/home/allen/hadoop-2.7.1/share/hadoop/common/*:/home/allen/hadoop-2.7.1/share/hadoop/hdfs:/home/allen/hadoop-2.7.1/share/hadoop/hdfs/lib/*:/home/allen/hadoop-2.7.1/share/hadoop/hdfs/*:/home/allen/hadoop-2.7.1/share/hadoop/yarn/lib/*:/home/allen/hadoop-2.7.1/share/hadoop/yarn/*:/home/allen/hadoop-2.7.1/share/hadoop/mapreduce/lib/*:/home/allen/hadoop-2.7.1/share/hadoop/mapreduce/*:/contrib/capacity-scheduler/*:/home/allen/hadoop-2.7.1/share/hadoop/yarn/*:/home/allen/hadoop-2.7.1/share/hadoop/yarn/lib/*
[allen@cmaster0 hadoop-2.7.1]$ bin/yarn daemonlog -getlevel cslave2:8042 NodeManager
Connecting to http://cslave2:8042/logLevel?log=NodeManager
Submitted Log Name: NodeManager
Log Class: org.apache.commons.logging.impl.Log4JLogger
Effective level: ERROR
[allen@cmaster0 hadoop-2.7.1]$
```

version命令

classpath命令

logs命令

笔者集群日志级别已被调为 ERROR

图 5-30 其他常见 YARN 命令

5.5 实战 YARN 编程

就像 Win7 上可以运行 QQ 和 Word 一样，YARN 上也可以运行不同类型的 Yarn-App (Yarn 应用程序)。不过，既然已经有了 Win7 那又为何出现了 YARN，最根本的原因是应用程序类型是不同的，YARN 上运行的一般都是分布式程序，而 Win7 上运行的大多为单机程序。

显然，编写分布式程序和编写单机程序应该不同，本节先讲述编写分布式程序时常见的并行化范式，然后讲述 Yarn-App 编写步骤，稍后的 5.6、5.7 则是编程实例。

5.5.1 常见并行化范式

当需要把一个应用程序并行化时，要么数据并行化，要么算法并行化，当然也可以两者结合，本节主要讲述最常见的几大并行化范式，即 M 范式、MSP 范式和 BSP 范式。

1. 并行化范式概述

(1) 数据并行化

数据并行化指的是对一个大文件，让每台机器（实际上是机器上的进程或线程）处理整个文件的一部分。比如对于一个 100G 的“大文件”，可以让 50 台服务器同时处理，这样，每台服务器只处理 2G 数据，这显然能够大大加快处理速度。常见的数据并行化范式为 M 范式、M-S-R 范式。

(2) 算法并行化

算法并行化指的是将整个程序算法部分拆分成多个独立模块，各模块并行执行，需要通信时各自通信。有多线程开发经验的读者应当能够明白，各个线程之间实际上是并行执行（CPU 多核），线程之间的同步与互斥一般通过共享变量实现，同理，将线程看成运行在不同机器上的进程，当进程之间须同步或互斥时通过 ZooKeeper 来协调进程间协作，BSP 模型的一个独立超步和超步之间就是通过 ZooKeeper 实现的。总之，即使是紧耦合类型的机器学习算法，利用 ZooKeeper 也可将其并行化，常见的算法并行化范式为 BSP 范式和 MPI 范式。

不过由于不同机器上进程启动的时空开销相对单机上的独立进程大出许多，加之进程间通信的时空开销，并行化反而使得算法执行代价增大（执行时间变长，占用资源更多）。当然，此时一定存在一个临界点，即当数据量（或算法复杂度）小于这个临界点时，单机执行得更快，当数据量（或算法复杂度）大于这个临界点时，并行化执行得更快。

(3) 先并行化后线程化

值得称道的是，在并行化过程中，可以将单机上的进程进一步线程化（Spark、Storm 皆采用该方式），此时分布式程序执行能力将得到进一步飞跃。比如编者有一机器学习任务 Kmeans，编者可以选择单机方式执行该程序；可以选择单机多线程方式执行该程序；可以选择将该程序均匀分给四个 slave，各 slave 内部以单机方式执行；更可以将该程序均匀分给四个 slave，各 slave 内部以多线程方式执行。最后的这种方式，即是编者要讲述的方式。

先将该任务分配到四个 slave 上，让这四个 slave 各启动一个进程 zSlave 并行计算，而在 zSlave 内部，比如 cs1ave1 上的 zSlave 进程内又用多线程的方法，让该 zSlave 并行执行，从而让并行化达到极致（图 5-31）。

“先并行化后线程化”属于并行化调优部分，不是本书重点，编者只在此处讲解，其他章节不再讲述。

2. M 范式

M 范式指的是，将全部数据等分成 n 份，各机器（进程）独立处理一份，完成处理后，再将处理结果合并到一起（图 5-32）。比如有个 40G 的大文件，可选择 4 台机器，让每台机器分别处理 10G。显然，各机处理时相互并不干扰，是并行的。

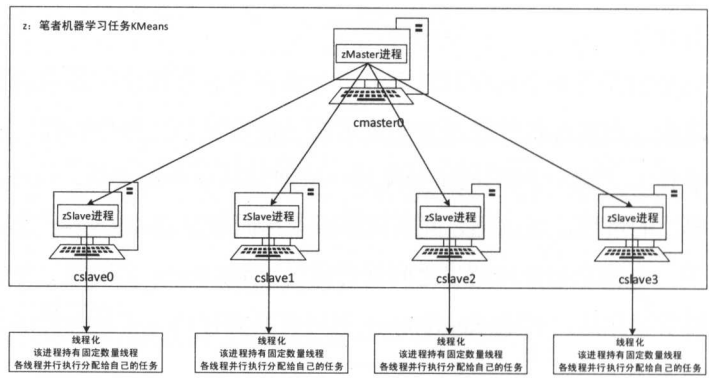


图 5-31 先并行化，后线程化

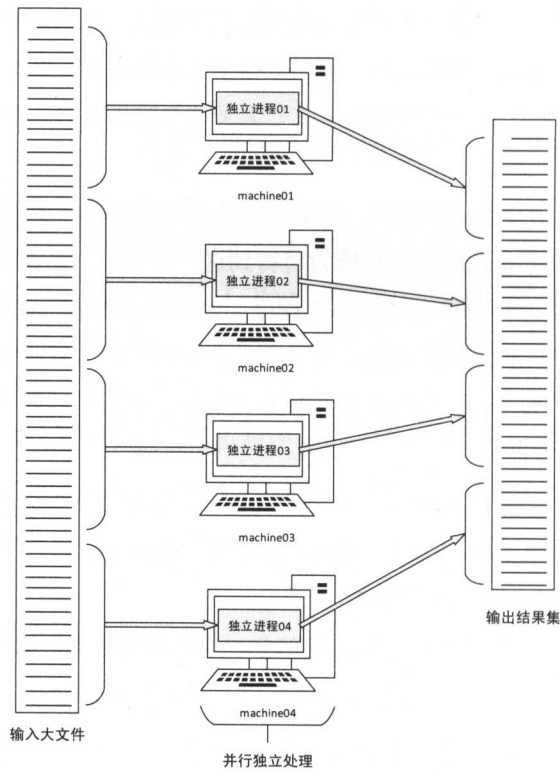


图 5-32 M 范式示意图

事实上，如果原始输入数据存储在 HDFS 上，那么各机读取数据时也是并行的，处理完后，各机器存储结果的过程也是并行的。需要注意的是，M 范式并非一定要处理数据，也可只是各机器上的进程申请一定量资源并启动进程，然后进程内进行大量 CPU 密集型操作。此外 M 范式还规定，各机器上的独立进程间无任何依赖，这点非常重要，如果进程间有依赖关系则是 BSP 范式，如果还要对结果集进行再处理（进程间无依赖），则是 M-S-R 范式。

作为最简单的并行化范式，M 范式应用反而很少，这主要是因为现实业务中，要么第一层处理进程（图中进程 01~04）之间有联系，要么还需要对第一层处理结果再处理，这两个需求分别对应 BSP 范式和 M-S-R 范式。不过作为最基础范式，还是希望读者熟练掌握，下一节将要讲述的 DistributedShell 是 M 范式的经典范例。

此外由于 DistributedShell 还是 YARN 编程的入门级示例，故可以说 M 范式是整个 YARN 编程基础中的基础。

3. M-S-R 范式

M-S-R（Map-Shuffle-Reduce）即著名的 MapReduce 模型，它是并行化最常见的范式，许多模型都可拆分成 M-S-R 范式，它的处理过程是：对于一个巨大输入文件，各机（图 5-33 中进程 01~04）分别处理一部分，待本机处理完后；该机上的处理进程会将处理结果按 Shuffle 规则发至下一层计算进程（图 5-33 中进程 06~08）；下一层计算进程将传递至本机的这些数据，进行再处理；最后，将计算结果输出（图 5-33）。

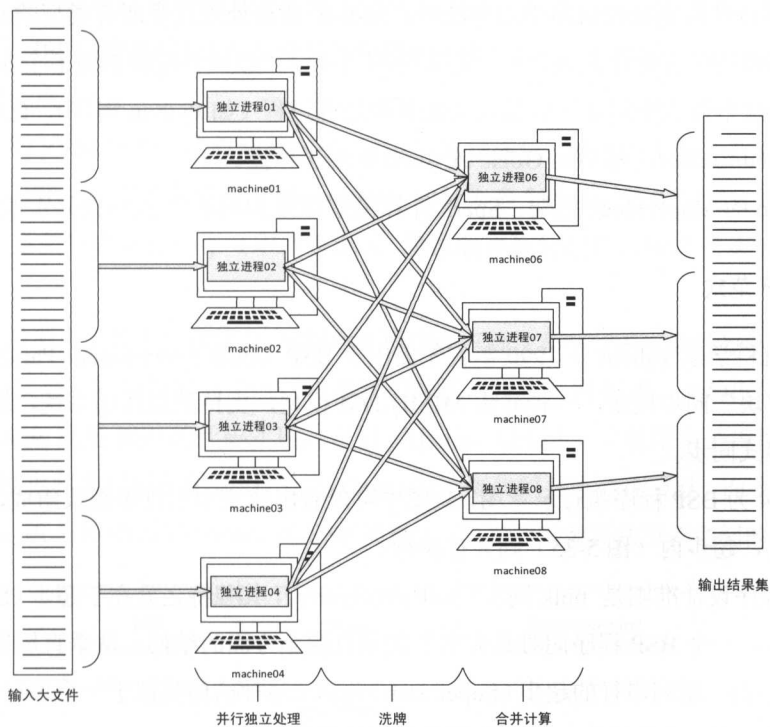


图 5-33 M-S-R 范式示意图

由于 Shuffle 规则的确定性，对于同类数据，按 Shuffle 规则，一定到同一个（下一层）计算进程（图 5-33 中进程 06~08），绝不可能出现同类数据发往不同进程的情况。同理，由于第一层处理进程（图 5-33 中进程 01~04）读到的数据可能隶属于各类数据，

故按 Shuffle 规则, 它的数据可能到任意一个下一层计算进程 (图 5-33 中进程 01~04)。

显然, 第一层处理进程 (图 5-33 中进程 01~04) 是并行执行的, 它们之间无任何依赖, 互不干扰。图中位于第一层和第二层之间的 Shuffle 过程也是并行的, 各机只关心本机要发送的数据, 和同一层的其他机器无任何关系。第二层的合并计算 (图 5-33 中进程 06~08) 也是并行的, 虽然进程 06 需要等待所有第一层进程结束后才可执行, 但这并不影响进程 06、07、08 之间的并行性。当输入数据存储在 HDFS 上时, 第一层处理进程读取数据的过程也是并行的, 当结果集存储在 HDFS 上时, 第二层存储过程也是并行的。

Hadoop 自带的 MapReduce 执行效率非常高, 按编者经验 (编者曾在上海互联网公司写了两年 MapReduce 程序), 2T 数据, 20 台服务器仅需要五分钟即可处理结束, 不过编者的 MR 程序里进行了大量优化, 故此处无可比性, 此处想强调的是 Hadoop 自带的 MapReduce 范式满足几乎所有需求, 值得读者深入研究。由于 MapReduce 是 Hadoop 生态圈支柱 (HDFS、YARN、MapReduce、BSP、ZooKeeper) 之一, 本书将在第 6 章重点讲述 MapReduce 范式。

除了耦合性特别高的机器学习算法外, M-S-R 适合处理几乎所有类型的数据分析任务, 它是最常用的并行化范式之一。在实际应用中, 一般业务逻辑都由多个 M-S-R 范式组成, 甚至这些范式中间还可以加入其他类型处理进程 (如 Python 程序), 此时可用 Tez 实现串式 MapReduce, 然后用 Oozie 或 Shell 组织工作流。

Hadoop 的 MapReduce 框架、Spark 计算模型都是 M-S-R 范式的具体实现。

4. BSP 范式

计算机科学家 Valiant 在 1990 年首先提出了 BSP (Bulk Synchronous Parallel) 模型, 它是一种“块”同步模型, 一种异步 MIMD-DM 模型, 支持消息传递系统, 块内异步并行, 块间显式同步。

图 5-34 为 BSP 程序执行示意图, 从图中可以看出该模型由许多超步组成, 各超步之间串行执行, 超步内 (图 5-35) 则并行执行。

BSP 程序设计准则是 bulk 同步 (bulk synchrony), 其独特之处在于超步 (Super Step) 概念的引入。一个 BSP 程序同时具有水平和垂直两个方面的结构, 从垂直层次上看, 一个 BSP 程序由一系列串行的超步 (Super Step) 组成, 这种结构类似于一个串程序结构。从水平上看, 在一个超步中, 所有的进程并行执行局部计算。

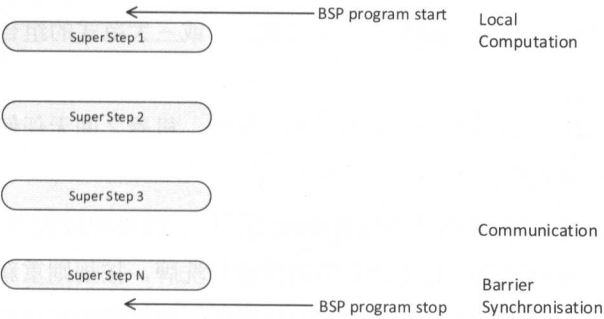


图 5-34 BSP 范式示意图

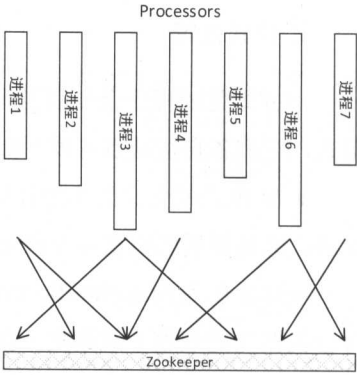


图 5-35 BSP 超步示例

BSP 应用由许多超步组成，而一个超步则分为如下三个阶段。

- 本地计算阶段：每个进程只处理本地数据。
- 全局通信阶段：对任何非本地数据进行处理。
- 栅栏同步阶段：等待所有通信行为结束。

相对于 MPI 和 PVM 这两种典型并行化模型（博士课程，难度较大），BSP 具有如下两个方面的优点。

● MPI 和 PVM 依赖于接收和发送的操作对，这样通信方式极易导致上层应用程序产生死锁，而 BSP 并行计算模型则是将一个程序划分为多个超步，我们知道，避免死锁的一大准则就是顺序地执行程序（申请资源），通过顺序执行超步，BSP 可大大减少死锁发生。

- BSP 模型由于其本身的特点，使得对于程序的正确性和时间的复杂性预测成为可能。

BSP 范式适合耦合性较高的统计机器学习算法，特别是图处理。大数据生态圈组件 Hama 和 Griaph 就是 BSP 的代码实现，它们（Hama、Griaph）主要用来处理增量计算和图计算，不过也并非只限于图处理，比如 Hama 的示例代码并不是图算法，而是 PageRank 和 K-Means，图 5-36 给出了 Hama、纯 BSP 类算法、深度学习类算法三者之间的层次关系。

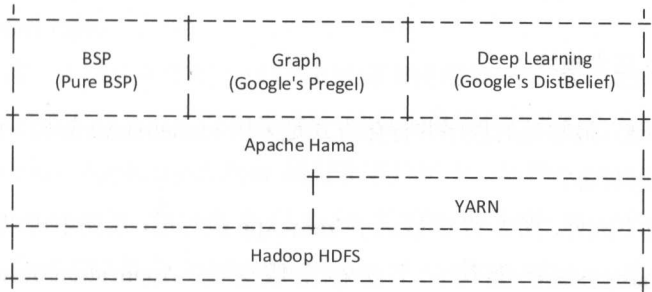


图 5-36 BSP 平台及其算法层次图

5. 范式比较

可并行化的程序至少应当满足 M、M-S-R、BSP 这三大范式之一或三大范式的组合范式，它们各自优缺点可罗列如下。

- M 范式的含义为本地计算，即每台机器处理整个数据的一部分，机器之间无任何依赖，对处理后的数据也不再进行任何处理。

- M-S-R 为 Map-Shuffle-Reduce 的缩写，即著名 MapReduce 模型，它的处理过程为，每台机器先处理整个数据集一部分，接着对第一轮处理后的数据进行洗牌，按规则重新组合，最后对重组后的数据进行第二轮处理。M-S-R 范式适合处理 I/O 密集型的松耦合作业，也可将 M-S-R 范式进行简单叠加，形成范式 $(M_1-S_1-R_1) \rightarrow (M_2-S_2-R_2) \cdots (M_n-S_n-R_n)$ ，不过本质上还是 M-S-R 范式。

- BSP 是 Bulk Synchronous Parallel 的缩写，中文翻译为“整体同步并行计算模型”，它的处理过程是，每台机器（实际上是进程）处理整个数据集一部分并根据自身处理进度先后进入阻塞状态（术语称 BarrierSynchronisation），待第一轮进程全部处理结束后，进入第二轮计算，一直循环往复，直至所有进程结束。BSP 范式适合处理 CPU 密集型的紧耦合作业，特别是机器学习型任务。

读者要注意，这三种范式可以组合使用，比如 M-S-R 自身组合 $(M_1-S_1-R_1) \rightarrow (M_2-S_2-R_2) \cdots (M_n-S_n-R_n)$ 或 M-S-R 和 BSP 的组合。

除了上述三大范式外，就像是 Win7 上可以运行 Tomcat 这类服务型应用一样，YARN 上也可以运行服务型应用程序，比如可以在 YARN 上运行 HBase。综上所述，YARN 上的应用程序大致可分为 M、M-S-R、BSP 和服务型这四大类型。

5.5.2 YARN 编程步骤

下面先讲解 Yarn-App 三大模块，接着分别讲述各模块编写步骤。

1. Yarn-App 三大模块

前文已经讲述，最常见的程序并行化方式是采用 master/slave 架构，让主进程指挥多个从进程并行执行任务。

Yarn-App 也不例外，图 5-37 即展示了 Yarn-App 执行层，从图中可以看出，Yarn-App 执行层也为 master/slave 架构，不妨称主服务为 ApplicationMaster，从服务为 ApplicationBusinessLogic。显然，仅包含 ApplicationMaster 和 ApplicationBusinessLogic 的 Yarn-App 是不够的，这是因为用户还要编写 ApplicationClient 向 YARN 提交任务，试

问，连任务都不能提交，何谈执行。

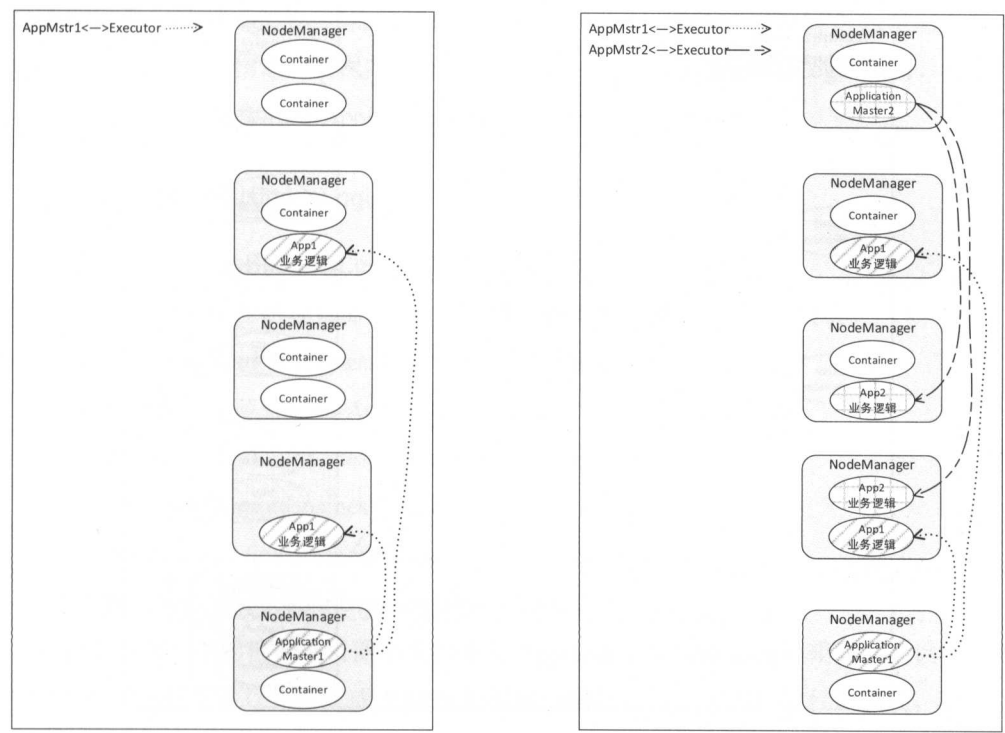


图 5-37 Yarn-App 执行层

综上所述，一个完整的 Yarn-App 应包含如下三个模块（图 5-38）：

- ApplicationBusinessLogic
- ApplicationClient
- ApplicationMaster

其中 ApplicationBusinessLogic 为应用程序的业务逻辑模块，ApplicationClient 为应用程序客户端，负责提交和监管（如查看、取消）应用程序。ApplicationMaster 负责控制整个应用程序运行，它是应用程序的并行化指挥地，需要指挥所有 Container 并行执行 ApplicationBusinessLogic。

为深入理解这三个模块的作用，可结合 RM 和 NM 细致分析 Yarn-App 整个生命周期。图 5-38 为在 YARN 上执行 App1 和 App2 时，这五大模块之间的角色关系。

由图 5-38 所示，ApplicationClient 运行在客户机上，其持有 ApplicationMaster 和 ApplicationBusinessLogic 模块。当客户要向 YARN 集群提交一个应用程序时，ApplicationClient 会向 RM 提交 ApplicationMaster 和 ApplicationBusinessLogic。在成功提交应用后，ApplicationClient 还提供了查询应用程序状态、终止应用程序执行等功能。

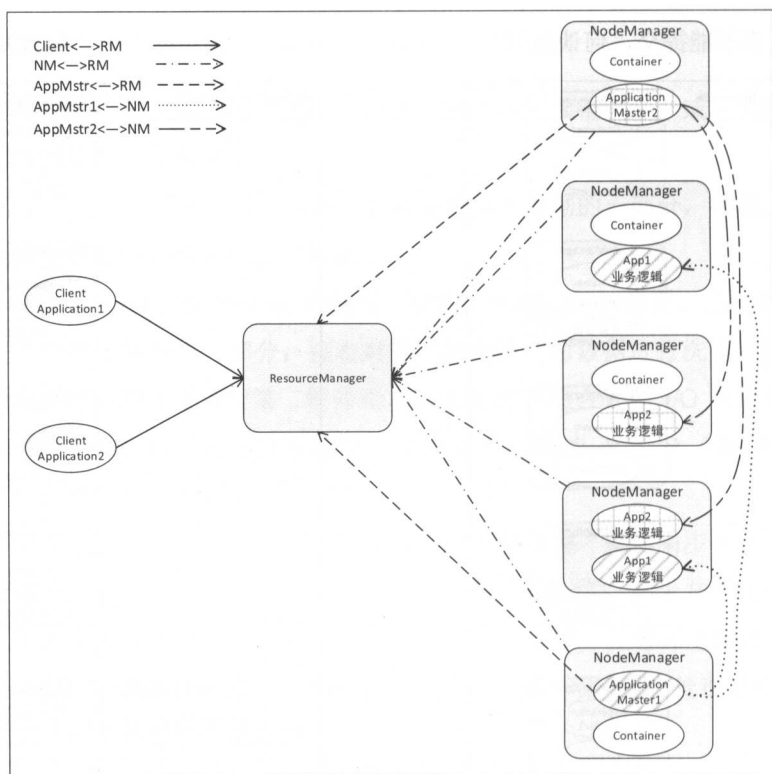


图 5-38 YARN 编程中涉及的实体

RM 根据 Client 提交的 ApplicationMaster, 选定一 Container 并在此 Container 上拉起此 ApplicationMaster。

拉起后的 `ApplicationMaster` 需要和 `RM` 通信申请资源（以 `Container` 形式），和 `NodeManager` 通信启动本 `ApplicationMaster` 拥有的 `Container` 并监控这些 `Container` 的运行状态。

当 ApplicationMaster 启动 Container 时，ApplicationMaster 会将应用程序的业务逻辑模块（即 ApplicationBusinessLogic）下沉到该 Container 里执行。

通过 YARN 交互可以看出,这三大模块缺一不可,至此,我们已成功地实现了将普通应用程序运行到 YARN 上。不过,在实际开发中,既可以将 ApplicationBusinessLogic 开发成一个独立模块,也可将 ApplicationBusinessLogic 直接加入 ApplicationMaster。但不管是单独还是加入 ApplicationMaster,在执行 Container 时,都必须将 ApplicationBusinessLogic 下沉到 Container 里,它才是整个应用程序的真正业务逻辑所在地。

既然一个 Yarn-App 共包含 ApplicationBusinessLogic、ApplicationClient 和 ApplicationMaster 三个独立模块，那么编写 Yarn-App 也就是编写这三个模块。不过，由于 ApplicationClient 和 ApplicationMaster 编写难度较大，大多数公司都是指派本公司优秀工程

师根据本公司业务逻辑开发通用 `ApplicationClient` 和 `ApplicationMaster` 并给出 `ApplicationBusinessLogic` 编程范式和接口。这样在开发 YARN 应用时，只根据本公司 `ApplicationMaster` 中指定的编程范式和接口，开发 `ApplicationBusinessLogic` 即可，在执行应用程序时，直接使用通用 `ApplicationClient` 和 `ApplicationMaster`。

2. ApplicationBusinessLogic

开发 `ApplicationBusinessLogic` 模块主要包含如下七大步骤：

- ①依据本部门在研项目需求书，开发 `ApplicationBusinessLogic` 模块。
- ②判断 `ApplicationBusinessLogic` 所属并行化范式。
- ③根据并行化范式，将 `ApplicationBusinessLogic` 改写成该范式。
- ④罗列 `ApplicationBusinessLogic` 执行时 Resource 清单。
- ⑤罗列 `ApplicationBusinessLogic` 执行时偏好清单。
- ⑥罗列 `ApplicationBusinessLogic` 执行时环境清单。
- ⑦编写执行 `ApplicationBusinessLogic` 的 Shell 脚本。

对于步骤①中的根据项目需求文档开发 `ApplicationBusinessLogic` 模块，请读者在开发时不要考虑任何并行化范式或 YARN 框架接口问题，只关注本模块代码实现，且可以使用任何编程语言。YARN 应用程序的 `ApplicationBusinessLogic` 支持各类语言，不再仅限于 Java，这主要是由于执行 `ApplicationBusinessLogic` 的 Container 直接运行在底层 OS 上并和 OS 交互，故只要 OS 支持此语言，`ApplicationBusinessLogic` 就可使用该语言编写。

目前常见的并行化范式主要分为 M 范式、M-P-R 范式、BSP 范式和服务型应用程序（M 范式特例）。M 范式的含义为本地计算；M-P-R 为 Map-Shuffle-Reduce 的缩写，即著名 MapReduce 模型，它适合处理 I/O 密集型的松耦合作业；BSP 是 Bulk Synchronous Parallel 的缩写，中文翻译为“整体同步并行计算模型”，它适合处理 CPU 密集型的紧耦合作业。开发工程师应根据业务逻辑，结合各范式划分标准，准确判断出在研应用程序所属类型。

想要执行一个程序时，至少要有程序的二进制代码（以及代码里依赖的其他代码，术语称依赖包），假如程序指定了输入输出，那还必须给它输入输出，此外若想要 OS 执行此程序，还必须配置 OS 相关工具环境（术语称环境变量）。最后，在执行程序时，直接使用 Shell 命令执行即可，如下述命令完成在 `iclient0` 上使用 java 命令启动 WordCount，该程序用来计算文件夹 `in` 下面的文件，并将结果存至 `out` 文件。

```
[allen@iclient0 ~]$ java -Xmx1024M -cp ds.jar njupt.WordCount /user/allen/in /user/allen/out
```

此处，jar 包 `ds.jar`、内存大小 1G、输入文件 `in`、输出 `out` 为程序对应的资源清单，`iclient0` 机的 Java 安装路径以及 Java 安装路径下的基础支持包（如 `dt.jar`、`tools.jar` 等）路

径为系统环境变量, 上述整条命令称为执行 ApplicationBusinessLogic 的 Shell 脚本。

ApplicationMaster 在将 ApplicationBusinessLogic 下沉至 Container 执行时, 需要使用此 Shell 命令, 执行 ApplicationBusinessLogic 代码。

3. ApplicationMaster

ApplicationMaster 模块的编写步骤大致如下:

- ①创建并启动 AppMstr 到 RM 的静态实例 AMRMClientAsync。
- ②创建并启动 AppMstr 到 NM 的静态实例 NMClientAsyncImpl。
- ③AppMstr 使用 AMRMClientAsync 到 RM 注册自己。
- ④AppMstr 使用 AMRMClientAsync 到 RM 申请 Containers。
- ⑤AppMstr 使用 AMRMClientAsync 将 AppBusinessLogic 下沉到 Container 里。
- ⑥AppMstr 使用 AMRMClientAsync 将其他资源下放到 Container 里。
- ⑦AppMstr 调用 NMClientAsyncImpl 启动 Container。
- ⑧AppMstr 调用 NMClientAsyncImpl 启动监控方法监控 Container。
- ⑨根据 ApplicationBusinessLogic 所属范式, 决定是否执行下一层 Container。
- ⑩AppMstr 使用 AMRMClientAsync 向 RM 汇报应用程序执行结束。

如前文所述, ApplicationMaster 需要分别和 RM、NM 通信, 在程序中这主要是通过 AMRMClientAsync 和 NMClientAsyncImpl 实现的。其中 AMRMClientAsync 主要实现了 ApplicationMaster↔ResourceManager 之间通信, NMClientAsyncImpl 主要实现了 ApplicationMaster↔NodeManager 之间通信。

ApplicationMaster 对象在自身方法区创建 AMRMClientAsync 实例后, 它首先会通过 AMRMClientAsync 到 RM 处注册自己。

几乎同一时刻 (通过线程完成), ApplicationMaster 对象会在自身方法区创建 NMClientAsyncImpl 实例并通过此实例获取和 NodeManager 通信。

待上述两个实例创建后, ApplicationMaster 会通过 AMRMClientAsync 向 ResourceManager 申请资源 (以 Container 形式)。一旦申请到资源, RM 会将 ApplicationBusinessLogic 及其相关资源、环境和启动 Shell (用来启动 ApplicationBusinessLogic 的 Shell 语句) 下沉至 Container 里并调用 NMClientAsyncImpl 启动此 Container。

此外 ApplicationMaster 会通过调用 NMClientAsyncImpl 监控 Container 执行状态, 在所有 Container 完成时调用 AMRMClientAsync 汇报任务完成, 结束自身生命周期。

在上述 ApplicationMaster 编写步骤中, 当 ApplicationMaster 要下沉至 Container 的 ApplicationBusinessLogic 属于不同范式时, 其第 9 步的编写步骤并不相同, 下面分别讲述。

(1) M 范式

当 ApplicationBusinessLogic 仅是 M 范式时, ApplicationMaster 只执行一层 Container, 且各个 Container 之间无需任何依赖关系 (图 5-39)。此时 ApplicationMaster 编写难度较低, 只申请一层 Container 并执行这一层次 Container 即可, 无须管理 Container 之间通信, 示例代码为 DistributedShell。

(2) M-S-R 范式

当 ApplicationBusinessLogic 属于 M-S-R 范式时, ApplicationMaster 须执行两层 Container, 且第二层 Container 的输入是第一层 Container 的输出 (图 5-40)。此时 ApplicationMaster 编写难度进一步加大, 不但要编写代码控制第一层 Container, 接着编写代码申请并执行第二层 Container, 还要编写代码来严格控制第一层 Container 到第二层 Container 的输出规则。也就是 ApplicationMaster 里既要有代码和 Map、Shuffle、Reduce 这三个模块相对应, 还要维护它们之间的相互关系, 显然, 这时的 ApplicationMaster 编程难度太大, 编者建议直接使用系统自带的 MapReduce 模块, 模板中的 MRAppMaster 就是 MapReduce 模板对应的 ApplicationMaster, 其控制整个 M-S-R 执行过程。

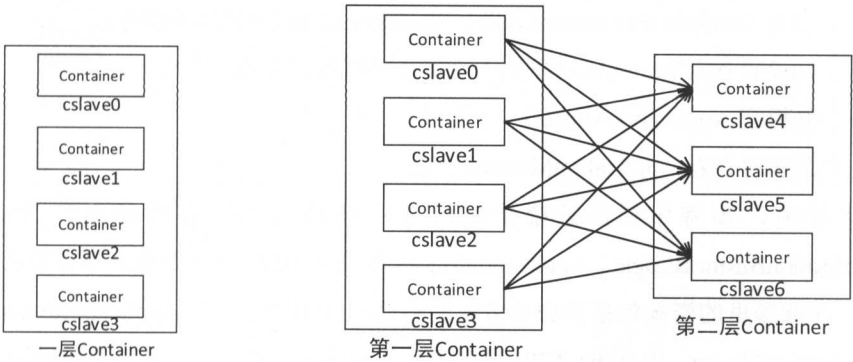


图 5-39 M 范式时 Container 示例图

图 5-40 M-S-R 范式时 Container 示例图

(3) BSP 范式

当 ApplicationBusinessLogic 属于 BSP 范式时, 各个进程间会出现同步或互斥关系, 此时 ApplicationMaster 需要执行多层 Container, 且不仅层次间 Container 有依赖关系, 同层内 Container 之间也有依赖关系。此时 ApplicationMaster 要完成所有层次的 Container 且还要控制层次间的同步操作 (可通过 ZooKeeper), 这样的 ApplicationMaster 编写难度极大, 编者建议使用 Apache 编写的 BSP 模板 Hama 或 Giraph。

综上所述, 编写一个兼具高容错性、高性能、通用 ApplicationMaster 是非常不容易的, 编者建议:

- 模仿 DistributedShell、Unmanaged-AM-Launcher、MRAappMaster、Giraph

- 直接使用通用 ApplicationMaster

一切的学习皆从模仿开始, 建议读者从 YARN 指定的三大示例 DistributedShell、Unmanaged-AM-Launcher、MRAappMaster 学起, 特别是 DistributedShell, 其自身的确没有提供实用功能, 但代码完整简单, 且能清晰阐述 YARN 应用程序编写过程, 建议读者模仿并调试这三大示例代码。

当前通用的 ApplicationMaster 主要是指 Unmanaged-AM-Launcher、MRAappMaster, 以及其他运行在 YARN 上的服务型框架 (如 Hive、HBase、Spark、Storm、REEF 等), 建议直接使用此类基于 YARN 的实用大数据组件, 无须独立开发。

4. ApplicationClient

ApplicationClient 模块的编写步骤大致如下:

- ①创建 YarnClient 实例。
- ②启动 YarnClient。
- ③创建 ApplicationMaster 上下文信息。
- ④将 ApplicationBusinessLogic、ApplicationMaster 扔至 HDFS。
- ⑤设置 ApplicationMaster 上下文信息 (资源, 环境, 命令)。
- ⑥提交 ApplicationMaster。
- ⑦实时查看 ApplicationMaster 状态。

其中, 步骤④和⑤是整个编程过程的核心点, 在步骤④, Client 需要将 ApplicationBusinessLogic、ApplicationMaster 提交至 HDFS (或其他共享存储系统)。

步骤⑤里的资源包括步骤④所做的存储在 HDFS 上的 ApplicationBusinessLogic、ApplicationMaster、内存量、CPU 核数等, 环境指的是执行此 ApplicationMaster 时应当设置的环境变量, 命令指的是使用什么命令来执行此 ApplicationMaster。

5. 小结

理论上, Yarn-App 包含 ApplicationBusinessLogic、ApplicationMaster 和 AppClient 三部分 (图 5-41), 故, 编写 Yarn-App 时可参考下述步骤。

Step1 程序员应先思考“此次项目”可如何并行 (M 范式、M-S-R 范式、BSP 范式)。

Step2 接着根据 ApplicationBusinessLogic 所属范式, 编写不同类型的 ApplicationBusinessLogic, 该部分实质上就是并行化时的用户程序。

Step3 接着根据 ApplicationBusinessLogic 所属类型, 编写 ApplicationMaster, 该模块一方面要向 RM 申请 Container 并将 ApplicationBusinessLogic 下沉至这些 Container 里执行, 另一方面还要控制整个范式执行流, 比如有的只有一层 Container (图 5-39), 有的

则为两层（图 5-40），有的为 N 层，有的甚至层内 Container 之间还有联系。实质上，ApplicationMaster 为范式和 YARN 接口的统一体，和程序员要并行的程序无关。

Step4 最后编写 ApplicationClient，该部分须持有 ApplicationBusinessLogic 和 ApplicationMaster（AppClient 有 AppBusinessLogic 和 AppMaster 的存储地址即为持有），ApplicationClient 须首先向 RM 提交本 Yarn-App，接着向 RM 申请第一个 Container（一般称 Container0）来执行 ApplicationMaster，当 ApplicationMaster 启动后，程序执行过程已和 ApplicationClient 无关，ApplicationMaster 会负责调度所有 Container 来并行执行 ApplicationBusinessLogic。

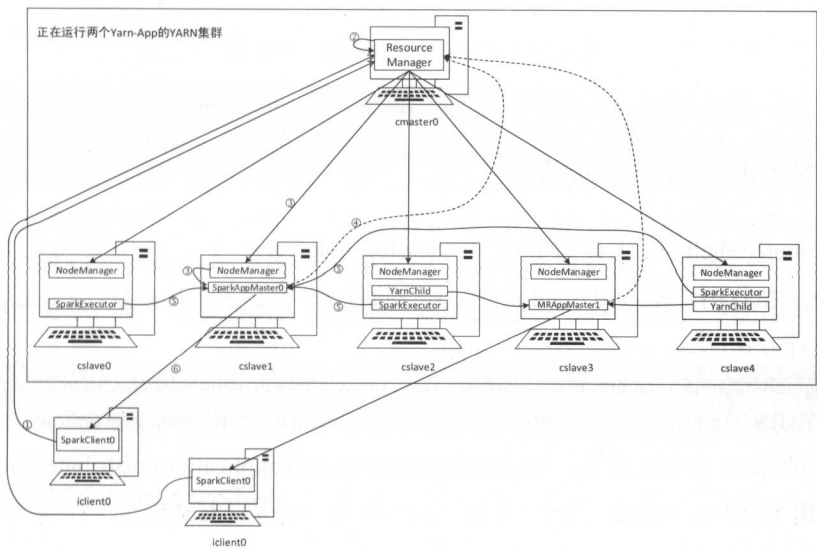


图 5-41 Yarn-App 执行过程

显然，即使是 M 范式时 ApplicationMaster 都有一定的编写难度，而 ApplicationClient 则非常通用，故实际开发时，一般直接使用通用 ApplicationMaster 和 ApplicationClient，程序员只开发 ApplicationBusinessLogic，各范式当前经典代码实现罗列如下：

- M 范式：DistributedShell。
- M-S-R 范式：MapReduce 框架、Spark 框架。
- BSP 范式：Giraph 框架。

故程序员在将“当前项目”并行化时，可使用现有的并行框架。

5.6 实战 YARN 编程之 DistributedShell

在学习新语言时, 第一个程序一般都是经典的 HelloWorld, DistributedShell 就相当于 YARN 的 HelloWorld。正如上节所述, DistributedShell 的确没有实质功能 (只是在各 Container 上执行了一个 Shell 命令), 但它简单完整, 能够非常清晰地讲述 YARN 应用程序编写过程, 几乎是学习 YARN 编程的必由之路。本节即分析 DistributedShell 的编写过程。

5.6.1 DistributedShell 简介

DistributedShell 默认存放在目录“/usr/localhdp/yarn/”下, 其主要功能是在每个 Container 上执行用户输入的 Shell 命令, 具体执行时, 有诸多可选参数, 下面给出两个最常用的执行方式。

```
[allen@iclient0 ~]$ yarn org.apache.hadoop.yarn.applications.distributedshell.Client \
-jar $YARN_DS/hadoop-yarn-applications-distributedshell.jar -shell_command uptime
```

这里的 yarn 为启动命令, “org.apache.hadoop.yarn.applications.distributedshell.Client”为本应用程序的 Client 类, 参数“-jar”指定 ApplicationMaster 存储位置, 即 hadoop-yarn-applications-distributedshell.jar 必须存在本应用程序的 ApplicationMaster 类, 参数“-shell_command”指定本应用程序的 ApplicationBusinessLogic, 即 Container 里真正执行的代码 (ApplicationMaster 自身所在 Container 除外)。

执行完后, 用户可到“\$HADOOP_HOME/logs/userlog/AppID/ContainerId”下, 查看 Shell 命令的输出。

上述命令为在 iclient0 上以 allen 用户执行 DistributedShell, 此时 DistributedShell 只会启动一个 Container 来运行 allen 指定的 Shell 命令 uptime。和上述命令对应, 下面的命令则要求 YARN 集群启动 3 个 Container 来执行用户 allen 给出的 Shell 命令 hostname (图 5-42)。

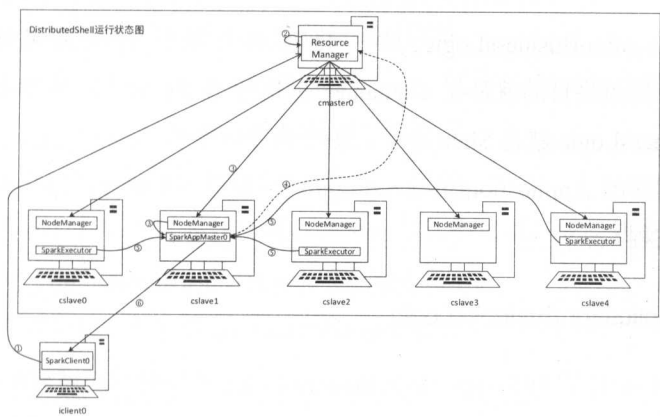


图 5-42 DistributedShell 执行状态图

```
[allen@iclient0 ~]$ yarn org.apache.hadoop.yarn.applications.distributedshell.Client \
-jar $YARN_DS/hadoop-yarn-applications-distributedshell.jar -shell_command hostname \
-num_containers 3
```

当然读者也可以把“3”改成“13”，这样，唯一不同的是，此时的 ApplicationMaster 会向 RM 申请 13 个 Container 来执行 Shell 命令。上述两个执行方式为调试 DistributedShell 时最常用的命令，至于其他参数，请读者参考帮助文档，自行练习，下面讲述该程序编写过程。

5.6.2 编写 DistributedShell

DistributedShell 内部包含如下三个类：

- Client
- ApplicationMaster
- DSConstants

类 Client 负责和 RM 交互，提交 ApplicationMaster 和 Shell 命令，在成功提交应用程序后，还可通过 Client 查看应用程序执行状态。类 ApplicationMaster 为整个应用程序控制中心，编写难度较高，主要负责向 RM 申请 Container，在申请到资源后和 NM 通信，启动 Container。类 DSConstants 中规定了一些 YARN 环境常量，可不必关心此类。

YARN 应用程序包含三部分，它和 DistributedShell 的对应见表 5-2。

表 5-2 DistributedShell 和 Yarn-App 三大模块对应关系表

YARN 应用程序标准模块	DistributedShell 模块
ApplicationBusinessLogic	Shell 命令
ApplicationClient	Client
ApplicationMaster	ApplicationMaster

除了 ApplicationBusinessLogic 外, 其他两个模块对应关系明显。实际上 DistributedShell 的最终目的就是在 Container 里执行 Shell 命令, 故 DistributedShell 的 ApplicationBusinessLogic 就是 Shell 命令。提交命令后参数 “-shell_command uptime” 指的就是本应用程序的 ApplicationBusinessLogic。下面分三个部分, 以实例讲述 Yarn-App 三大模块编写过程。

1. 编写 ApplicationBusinessLogic

编写 YARN 应用程序的 ApplicationBusinessLogic 模块时, 应遵循如下步骤:

- ①依据本部门在研项目需求书, 开发 ApplicationBusinessLogic 模块。
- ②判断 ApplicationBusinessLogic 所属并行化范式。
- ③根据并行化范式, 将 ApplicationBusinessLogic 改写成该范式。
- ④罗列 ApplicationBusinessLogic 执行时 Resource 清单。
- ⑤罗列 ApplicationBusinessLogic 执行时偏好清单。
- ⑥罗列 ApplicationBusinessLogic 执行时环境清单。
- ⑦编写执行 ApplicationBusinessLogic 的 Shell 脚本。

依据上述步骤, 下面一步步讲述。

- ①依据本部门在研项目需求书, 开发 ApplicationBusinessLogic 模块。

由于 DistributedShell 的 ApplicationBusinessLogic 是一个 Shell 命令, 而 Linux 系统本身已经提供大量 Shell 命令且可在命令行下正确执行, 故不用单独开发此模块。

- ②判断 ApplicationBusinessLogic 所属并行化范式。

明显, 每个 Container 启动一个 Shell 并执行此 Shell, NodeManager 会将各个 Container (也即 Shell) 执行的输出送至 NodeManager 所在机以 ContainerID 命令的标准输出日志, 而各个 Container 之间也无任何依赖关系, 故 DistributedShell 的 ApplicationBusinessLogic (即 Shell 命令) 属于 M 范式。

- ③根据并行化范式, 将 ApplicationBusinessLogic 改写成该范式。

在 M 范式的应用程序里, 每个 Container 都要执行完整的 ApplicationBusinessLogic, 故无须任何并行化改写, 直接执行即可。

- ④罗列 ApplicationBusinessLogic 执行时 Resource 清单。

在 Container 上执行 Shell 命令时, 所需的资源一般是 CPU、内存和输出目录、ApplicationBusinessLogic 代码 (Shell 命令), 故 Container 上执行 AppBusinessLogic 资源清单为: ApplicationBusinessLogic 代码—Shell 命令 (一般存储在 HDFS 上)、CPU 个数、内存量和输出目录。

⑤罗列 ApplicationBusinessLogic 执行时偏好清单。

由于 DistributedShell 的 ApplicationBusinessLogic 只是在 Container 里执行一个 Shell，并不会处理数据，无任何输入，基本无任何偏好可设置。

⑥罗列 ApplicationBusinessLogic 执行时环境清单。

Shell 命令在执行时只依赖系统环境，并不需要其他设置，故此处也无须设置。

⑦编写执行 ApplicationBusinessLogic 的 Shell 脚本。

Shell 命令直接即可，无须编写，NodeManager 会自动和此层 OS 交互，启动此 Shell，故此处即为 Shell 命令本身。

2. 编写 ApplicationClient

在编写 ApplicationClient 模块时，应当遵循如下步骤：

- ①创建 YarnClient 实例。
- ②启动 YarnClient。
- ③创建 ApplicationMaster 上下文信息。
- ④将 ApplicationBusinessLogic、ApplicationMaster 扔至 HDFS。
- ⑤设置 ApplicationMaster 上下文信息（资源，环境，命令）。
- ⑥提交 ApplicationMaster。
- ⑦实时查看 ApplicationMaster 状态。

下面按上述步骤，分析 DistributedShell 的 Client 模块，不过由于 Client 源码太多，下述代码中，编者只解释了最关键的执行步骤，其他代码（除了设置令牌权限外）几乎都是旁枝末节，读者可不必关心。

```
/**
 *YARN 编程示例代码 DistributedShell，由 ASF 编写，Allen（叶晓江）加注并整理
 *Client 类为 DistributedShell 三大模块的 ApplicationClient 模块，负责和 RM 交互提交整个应用程序
 *Client 主要向 RM 提交启动 ApplicationMaster 的 Container 上下文信息（jar 包、环境、命令等）
 *此外，Client 还要将 ApplicationBusinessLogic（此处为 shell 命令）拷至 HDFS（以供后续进程 ApplicationMaster 执行时调用）
 *在成功提交应用程序后，还可通过 Client 查看本应用执行过程
 */
public class Client {
    //Client 端整个流程
    public boolean run(){
        YarnClient yarnClient = YarnClient.createYarnClient();//创建 YarnClient 实例
        yarnClient.start();//启动 yarnClient 实例
        YarnClientApplication app = yarnClient.createApplication();//申请创建 YARN 应用程序
        //设置应用程序上下文信息
```

```

ApplicationSubmissionContext appContext = app.getApplicationSubmissionContext();
ApplicationId appId = appContext.getApplicationId();//获取 YARN 分配给本应用程序的 AppID
appContext.setApplicationName(appName);//个性化设置本应用程序名
//声明一个集合变量, 用户存储 ApplicationMaster 对应的部分上下文信息
Map<String, LocalResource> localResources = new HashMap<String, LocalResource>();
FileSystem fs = FileSystem.get(conf);//获取 HDFS 引用
//将 ApplicationMaster 拷贝至 HDFS
addToLocalResources(fs, appMasterJar, appMasterJarPath,appId.toString(), localResources, null);
//将 ApplicationBusinessLogic (也即 Shell 命令) 拷贝至 HDFS
addToLocalResources(fs, null, shellCommandPath, appId.toString(),localResources, shellCommand);
//声明一个集合变量, 用来存储执行 ApplicationMaster 时的环境变量
Map<String, String> env = new HashMap<String, String>();
env.put("CLASSPATH", classPathEnv.toString());//向集合中添加环境举例
//设置 ApplicationMaster 对应的 Container 上下文信息
ContainerLaunchContext amContainer = ContainerLaunchContext.newInstance(localResources,
env,commands, null, null, null);
Resource capability = Resource.newInstance(amMemory, amVCores);//实例化内存和 CPU 资源
appContext.setResource(capability);//设置本应用程序申请的内存和 CPU 资源
//指定执行本应用程序的 ApplicationMaster 的 Container 为 amContainer
appContext.setAMContainerSpec(amContainer);
Priority pri = Priority.newInstance(amPriority);//获取系统支持优先级信息
appContext.setPriority(pri);//设置本应用程序优先级
appContext.setQueue(amQueue);//设置本应用程序所属队列
yarnClient.submitApplication(appContext);//提交本应用程序
ApplicationReport report = yarnClient.getApplicationReport(appId);//获取本应用程序当前执
行进度
if(执行顺利){yarnClient.killApplication(appId);}//如果执行不顺利, 可直接终止本应用程序
YarnApplicationState state = report.getYarnApplicationState();//获取本应用程序执行结果信息
}
}

```

在了解 Client 整体编写流程后, 请读者务必一句一句调试源代码。

3. 编写 ApplicationMaster

在编写 ApplicationMaster 时, 应遵循如下步骤:

- ①创建并启动 AppMstr 到 RM 的静态实例 AMRMClientAsync。
- ②创建并启动 AppMstr 到 NM 的静态实例 NMClientAsyncImpl。
- ③AppMstr 使用 AMRMClientAsync 到 RM 注册自己。
- ④AppMstr 使用 AMRMClientAsync 到 RM 申请 Containers。
- ⑤AppMstr 使用 AMRMClientAsync 将 AppBusinessLogic 下沉到 Container 里。
- ⑥AppMstr 使用 AMRMClientAsync 将其他资源下放到 Container 里。

- ⑦AppMstr 调用 NMClientAsyncImpl 启动 Container。
- ⑧AppMstr 调用 NMClientAsyncImpl 启动监控方法监控 Container。
- ⑨根据 ApplicationBusinessLogic 所属范式，决定是否执行下一层 Container。
- ⑩AppMstr 使用 AMRMClientAsync 向 RM 汇报应用程序执行结束。

由于 DistributedShell 属于 M 范式，故第 9 步实际上是不需要的，ApplicationMaster 只申请一层 Container（且 Container 之间无依赖）并执行完这层 Container 即可，无需第二层 Container。

不过，即使是这种最基本的 M 范式，ApplicationMaster 编写难度依旧很大，在实例 AMRMClientAsync 和 NMClientAsyncImpl 中都涉及大量的回调函数。ApplicationMaster 源码近 2000 行，编者选取了其中最主要部分并加注释如下：

```
/**
 *Write by ASF, Note by Allen（叶晓江）
 *ApplicationMaster 主要负责
 *   和 RM 通信申请 Container
 *   和 NM 通信下沉 Shell 至各 Container 里并启动 Container
 */
public class ApplicationMaster {
    //ApplicationMaster 执行总流程
    public void run(){
        //创建 AMRMClientAsync.CallbackHandler 回调
        AMRMClientAsync.CallbackHandler allocListener = new RMCallbackHandler();
        //创建 AMRMClientAsync 实例
        AMRMClientAsync amRMClient = AMRMClientAsync.createAMRMClientAsync(1000,
allocListener);
        amRMClient.init(conf);//依据基本配置，初始化 amRMClient 实例
        amRMClient.start();//启动 amRMClient，amRMClient 为独立线程
        //创建 NMClientAsync.CallbackHandler 回调
        NMCallbackHandler containerListener = createNMCallbackHandler();
        //创建 NMClientAsync 实例
        NMClientAsync nmClientAsync = new NMClientAsyncImpl(containerListener);
        nmClientAsync.init(conf);//依据基本配置，初始化 nmClientAsync 实例
        nmClientAsync.start();//启动 nmClientAsync，nmClientAsync 为独立线程
        //使用 amRMClient，到 RM 处注册自己（本 ApplicationMaster）
        RegisterApplicationMasterResponse response =
amRMClient.registerApplicationMaster(appMasterHostname,
appMasterRpcPort,appMasterTrackingUrl);
        //依据本次应用程序指定的 Container 数量，逐个申请 Container 实例
        for (int i = 0; i < numTotalContainersToRequest; ++i) {
            Priority pri = Priority.newInstance(requestPriority);//实例化本预申请 Container 的优先级
            //实例化预申请 Container 的内存和 CPU 资源
            Resource capability = Resource.newInstance(containerMemory,containerVirtualCores);
```



```

//实例化预申请 Container 对象
ContainerRequest containerAsk = new ContainerRequest(capability, null, null, pri);
amRMClient.addContainerRequest(containerAsk);//
}
//ApplicationMaster 主流程结束，下沉 ApplicationBusinessLogic
//启动 RM 分配的 Container、监控 Container、汇报所有 Container 结束等操作，都在回调函数
里完成
}

/**
 *AMRMClientAsync.CallbackHandler 回调函数，ASF 编写，Allen 加注，本回调主要完成
 *1.所有 Container 完成时重置标志位
 *2.对 ApplicationMaster 申请到的每个 Container，启动独立线程执行此 Container
 */
private class RMCallbackHandler implements AMRMClientAsync.CallbackHandler {
    @Override //重载方法，所有 Container 都结束时执行，主要完成重置标志位
    public void onContainersCompleted(List<ContainerStatus> completedContainers) {}
    @Override //重载方法，非常重要，对本 ApplicationMaster 申请到的每个 Container，启动之
    public void onContainersAllocated(List<Container> allocatedContainers) {
        for (Container allocatedContainer : allocatedContainers) {
            //对本 ApplicationMaster 申请到的每个 Container，以独立线程，启动之
            LaunchContainerRunnable runnableLaunchContainer = new LaunchContainerRunnable(allocatedContainer,
containerListener);
            Thread launchThread = new Thread(runnableLaunchContainer);
            launchThread.start();//以独立线程，启动此 Container
        }
    }
    @Override //重载方法，ApplicationMaster 执行结束时执行
    public void onShutdownRequest() {done = true;}
    @Override //重载方法，NodeManager 发生改变时执行
    public void onNodesUpdated(List<NodeReport> updatedNodes) {}
    @Override //重载方法，下一次心跳时将会把当前执行进度信息传递至 RM
    public float getProgress() { //进度计算方式很简单，以完成 Container 数除以所有 Container 数
        return (float) numCompletedContainers.get() / numTotalContainers;
    }
    @Override //重载方法，发生错误时执行
    public void onError(Throwable e) {done = true;amRMClient.stop();}
}

/**
 *NMClientAsync.CallbackHandler 回调函数，Write by ASF,Note by Allen(叶晓江)，本回调主要
完成
 *监控此 NodeManager 上对应本次 ApplicationMaster 的 Container
 */
static class NMCallbackHandler implements NMClientAsync.CallbackHandler {

```

```

//记录本 NodeManager 上和此 ApplicationMaster 对应的所有 Container
private ConcurrentMap<ContainerId, Container> containers = new ConcurrentHashMap<
ContainerId, Container>();
private final ApplicationMaster applicationMaster;//本次应用程序的 ApplicationMaster
public NMCallbackHandler(ApplicationMaster applicationMaster) {
    this.applicationMaster = applicationMaster;}
public void addContainer(ContainerId containerId, Container container) {
    containers.putIfAbsent(containerId, container);}
@Override //重载方法, Container 结束时执行
public void onContainerStopped(ContainerId containerId) { containers.remove(containerId);}
@Override //重载方法, 查看 Container 状态
public void onContainerStatusReceived(ContainerId containerId, ContainerStatus containerStatus)
{}
@Override //重载方法, 查看 Container 状态
public void onContainerStarted(ContainerId containerId, Map<String, ByteBuffer> allServiceResponse) {
    Container container = containers.get(containerId);
    applicationMaster.nmClientAsync.getContainerStatusAsync(containerId,
container.getNodeId());}
@Override //重载方法, 启动出差错时发生
public void onStartContainerError(ContainerId containerId, Throwable t) {}
@Override //重载方法, Container 出差错时发生
public void onGetContainerStatusError(ContainerId containerId, Throwable t) {}
@Override //重载方法, 关闭 Container 出差错时发生
public void onStopContainerError(ContainerId containerId, Throwable t) {containers.remove
(containerId);}
}
/**
 * LaunchContainerRunnable 类, Write by ASF, Note by Allen(叶晓江), 本类主要完成启动
Container
 */
private class LaunchContainerRunnable implements Runnable {
    Container container;//关联单独 Container
    NMCallbackHandler containerListener;//关联本 Container 对应的 NMCallbackHandler
    public LaunchContainerRunnable(Container lcontainer, NMCallbackHandler containerListener) {
        this.container = lcontainer;//关联单独 Container
        this.containerListener = containerListener;//关联本 Container 对应的 NMCallbackHandler
    }
    @Override
    //重载方法, 非常重要, 将 ApplicationBussinessLogic 下沉至 Container 里并调用 NMClientAsyncImpl
    启动此 Container
    public void run() {
        //执行 ApplicationBussinessLogic 的 Container 的本地资源
        Map<String, LocalResource> localResources = new HashMap<String, LocalResource>();
        //本 Container 上下文信息

```



```
ContainerLaunchContext ctx = ContainerLaunchContext.newInstance(localResources, shellEnv,
shellCommand, null, allTokens.duplicate(), null);
    containerListener.addContainer(container.getId(), container);//侦听此 Container 执行状态
    nmClientAsync.startContainerAsync(container, ctx);//调用 NMClientAsyncImpl 启动此 Container
}
}
}
```

4. 小结

编写好 ApplicationBusinessLogic、ApplicationMaster 和 ApplicationClient 这三个模块后，将整个项目打成一个 Jar 包，显然，打到同一个包后，所谓的 ApplicationClient 持有 ApplicationMaster 和 ApplicationBusinessLogic 就是指 ApplicationMaster 是 AppClient 类的成员变量。执行时，可模仿下述命令：

```
[allen@iclient0 ~]$ yarn org.apache.hadoop.yarn.applications.distributedshell.Client \
-jar $YARN_DS/hadoop-yarn-applications-distributedshell.jar -shell_command hostname \
-num_containers 10
```

请读者调试并理解整个 DistributedShell 代码，它是 YARN 编程基础中的基础。

5.7 实战 YARN 编程之三大范式

在 Hadoop1.0 中，M-S-R 范式是系统提供的唯一编程模板。而在 Hadoop2.0 中，Hadoop 设计者将原集群资源管理、作业调度这两大功能上提，独立成 YARN 模块，负责管理整个集群资源和任务调度；将 M-S-R 范式的控制逻辑下沉至 MapReduce 框架里，负责执行 MR-App。

这种架构使得 Hadoop2.0 可支持各类应用程序，目前不但已经有诸多基于 YARN 的成熟框架，还有大量应用正在开发中，许多商业公司也在利用 YARN 平台优势，开发商业应用框架，可以说 YARN 为大数据平台带来了一次技术浪潮，不过对于个人来说，YARN 编程难度较大，此时最好参考通用大数据组件，表 5-3 为三大并行范式及其示例实现，本书后续章节将会逐一讲述。

表 5-3 并行化范式及其示例框架

并行范式	示例实现
M 范式	DistributedShell 框架
M-S-R 范式	MapReduce 框架
BSP 范式	Giraph 框架

5.7.1 DistributedShell

前文已经介绍了 DistributedShell 功能和开发过程，作为编写 Yarn-App 的示例代码，DistributedShell 最大的意义就是指导读者如何编写 Yarn-App。不难发现，DistributedShell 隶属于 M 范式，它也是该范式的示例程序，由于 M 范式是最简单的并行化范式，DistributedShell 可以说是最基本的 YARN 应用程序。

Yarn-App 一般包含三个标准模块，表 5-4 显示了 DistributedShell 中，这三个模块主类名，这对读者将有所帮助。

表 5-4 DistributedShell 对应的 Yarn-App 三大模块

YARN 应用程序标准模块	DistributedShell 框架对应类
ApplicationBussinessLogic	用户编写的 Shell 命令
ApplicationClient	Client.java
ApplicationMaster	ApplicationMaster.java

DistributedShell 功能是，在 Container 里运行 Shell 命令，在默认情况下 DistributedShell 只会启动一个 Container 并在里面执行 Shell 命令，不过可以通过参数指定启动多个 Container，当然，每个 Container 里实际上还是执行同一个 Shell 命令，有关 DistributedShell 的具体细节，请参考 5.6 节。

5.7.2 MapReduce

分布式处理的五大基石是 HDFS（分布式存储系统）、YARN（分布式资源管理系统）、MapReduce（并行计算模型）、BSP（并行计算模型）和 ZooKeeper（分布式锁服务，实质上由 Paxos 算法实现），诚然 MapReduce 适合处理全局而非增量数据，且互联网上的大部分数据都是增量的，需要的是实时增量的处理机制（请参见 DataFu、Flink 等组件）。但这并不代表 MapReduce 已被淘汰，相反，大量的上层组件（如 Hive、Pig）底层都依赖 MapReduce，甚至 DataFu 里的增量处理算法本身就是用 MapReduce 实现的（增加标记位），故 MapReduce 依旧是大数据处理方面首选模型，依旧是本书重点。

当前 littleCstor 上的 MapReduce 框架就是并行化范式 M-S-R 的具体实现，前面已经讲述，标准 YARN 应用程序应包含 ApplicationBusinessLogic、ApplicationClient、ApplicationMaster 这三个模块，同样，MapReduce 也应有这三个模块，表 5-5 即为这三个模块在 MapReduce 框架中的主类名：

表 5-5 MapReduce 框架主类

YARN 应用程序标准模块	MapReduce 框架对应类
ApplicationBusinessLogic	用户自定义 Mapper 类、Partition 类、和 Reduce 类
ApplicationClient	YARNRunner.java
ApplicationMaster	MRAPPMaster.java

作为 M-S-R 范式的示例应用框架，MapReduce 简单、高效，应用广阔，如果读者需要编写 M-S-R 类 ApplicationBusinessLogic，编者建议直接使用 MapReduce 框架，自己编写这一套框架几乎不可能。如果读者想学习 M-S-R 范式框架编写，建议研究 MapReduce 源码，注意是 Hadoop2.0 及其以后的 MapReduce 源码，Hadoop1.0 中不存在 YARN，由于 MapReduce 非常重要，编者将用第 6 章整章讲解 MapReduce。

5.7.3 Giraph

Giraph 是一个基于 YARN 的图处理框架，它是谷歌 Pregel 的开源实现，可以把 Giraph 和 Pregel 看成 BSP 算法的代码实现。目前，Giraph 广泛应用于 Facebook、Twitter 和 LinkedIn 等著名大公司。

需要注意的是，Giraph1.0 之前的版本并不支持 YARN，从 1.0 开始，Giraph 开始迁移到 YARN 上，作为 BSP 范式的实例实现，Giraph 也包含 YARN 标准应用程序的三大模块，其主类名及其对应关系见表 5-6。

表 5-6 Giraph 框架主类

YARN 应用程序标准模块	Giraph 框架对应类
ApplicationBusinessLogic	用户自定义 BasicComputation 类
ApplicationClient	GiraphYarnClient.java
ApplicationMaster	GiraphApplicationMaster.java

作为 BSP 范式的示例应用框架之一，Giraph 简单、高效，应用广阔，如果读者需要编写 BSP 类 ApplicationBusinessLogic，编者建议直接使用 Giraph 框架，自己编写这一套框架几乎不可能。如果读者想学习 BSP 范式框架编写，建议研究 Giraph 或 Hama 源码，注意是 Giraph1.0 后的源码，上面已经说明 Giraph1.0 以前的版本不支持 YARN。由于 littleCstor 上并未部署 Giraph，本书不再讲解 Giraph，有兴趣的读者请参阅“<http://giraph.apache.org>”。

习 题

1. 请从功能模块方面，简述单机 OS 和分布式 OS 的区别与联系。
2. 在集群环境下，为何需要一个统一的资源仲裁中心？
3. 简述 YARN 功能作用及其体系架构。
4. YARN 采用何种机制来抽象集群资源？
5. YARN 下有哪些调度策略？哪个策略能够实现公有云？
6. 简述手工部署 YARN、使用 Ambari 部署 YARN 的部署步骤。
7. 简述 YARN 访问接口。
8. 简述使用 Maven 时，YARN 开发环境搭建步骤，不使用 YARN 时又如何？
9. 简述常见的三大并行化范式以及基于 YARN 时，这三大并行化范式的编程步骤。
10. 在大型系统中，如何使用 YARN 来提供在线和离线服务？
11. 简述常见的基于 YARN 的编程框架。
12. 简述 YARN 的数据和服务安全机制。
13. 简述 YARN-App 执行步骤。

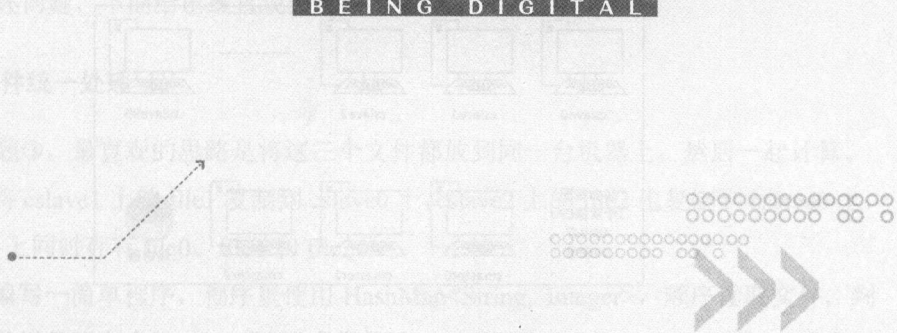
参考文献

- [1] https://en.wikipedia.org/wiki/System_software
- [2] <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/yarn.html>
- [3] <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/index.html>
- [4] <http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/WebServicesIntro.html>

第6章

分布式并行处理 MapReduce

HADOOP
BEING DIGITAL



M-S-R 范式是处理海量数据时最常用的并行模型，MapReduce 框架则是 M-S-R 范式的代码实现。本章以 M-S-R 范式为并行化模型，MapReduce 为系统框架，通过具体实例详细介绍 MapReduce 编程，使读者能够快速掌握编写 MapReduce 并行程序的思想和方法。

6.1 并行化范式 M-S-R 引例

大数据对存储和计算带来了巨大的挑战，下面给出一个场景和三类问题，请读者在此场景下讨论并解决这三类问题。

6.1.1 问题描述

现有一些配置完全相同的机器 cmaster0、cmaster1、cmaster2、cslave0~cslaveN，已知每台机器都是 1 个双核 CPU，5GB 硬盘，4G 内存（图 6-1），请按下述要求回答问题。

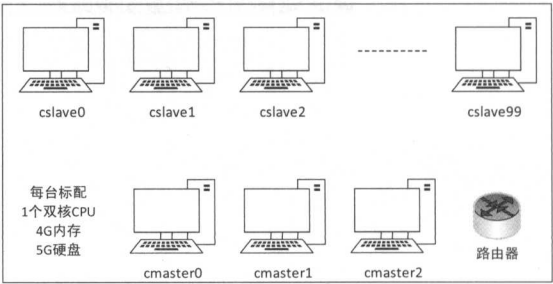


图 6-1 场景硬件配置图

1. 第一类问题：存储

问题①小文件统一存储：现有三个大小都是 1GB 的文件 file0、file1 和 file2，要求在任选机器的情况下，将这三个文件存入机器，并且对外显示时（逻辑上）这三个文件存于同一个存储空间。

问题②大文件独立存储：现有一个大小为 6G 的新文件 file9，要求存入机器后对外显示时，逻辑上依旧为一个完整文件。

2. 第二类问题：计算

问题③多文件统一处理：在问题①下，统计 file0、file1、file2 这三个文件里每个单词出现的次数。

问题④处理大文件：在问题②下，统计 file9 中，每个单词出现的次数。

3. 第三类问题：可靠性

问题⑤存储可靠性：假设存储 file0 的机器宕机了，问如何保证数据不丢失。

问题⑥计算可靠性：假如 cslave0 正在计算 file0，但在计算过程中 cslave0 突然宕机，问如何保证计算任务继续进行。

为求简单明了，上述场景与问题的描述可能不够完善，读者也暂不考虑诸如数据库、压缩存储、NFS 等方案，把思路放在分布式上，下面给出最直观解答。

由于第一类存储问题已经在 HDFS 章节讲解，这里不再赘述，只讲解计算和可靠性计算。

6.1.2 常规解决方案

针对上述问题，下面给出最直观的解决方案。

1. 多文件统一处理

对于问题③，最直观的思路是将这三个文件都放到同一台机器上，然后一起计算。

Step1 将 cslave1 上的 file1 复制到 cslave0 上，cslave2 上的 file2 也复制到 cslave0 上，这样 cslave0 上同时存有 file0、file1 和 file2。

Step2 编写一简单程序，程序里使用 `HashMap<String, Integer>`，顺序读取文件，判断新读取的单词是否存在于 `HashMap`，存在 `Integer+1`，不存在则 `HashMap` 里加入这个新单词，`Integer` 置为 1，记此程序为 `WordCount`。

Step3 将此程序 `WordCount` 放在 cslave0 上执行，得出结果。

统计单词个数问题已解决，如果现实问题真是如此，上述方案简单明了，易于操作，的确是个好方法。可是假如数据分布在 100 台机器上，每台机器存的不是 1GB 而是 1TB，仅仅数据复制这一步，就需要花去几天时间，并且普通服务器硬盘空间一般都是 2TB 到 4TB，很少有配置 100TB 硬盘的，大数据环境下已不可能依赖单台服务器存储和处理数据了。

2. 处理大文件

对于问题④, 由于单台机器已存不下 file9, 可以分两次处理, 然后将结果合到一起。

Step1 将 file9 分成大小相等的两个部分, 分别记 file9-b1 和 file9-b2, b 为 block 缩写。

Step2 编写 WordCount 程序。

Step3 在 cslave3 上先存储 file9-b1, 然后使用 WordCount 程序计算 file9-b1。

Step4 删除 file9-b1, 将 file9-b2 存于 cslave3, 然后使用 WordCount 程序计算新数据 file9-b2。

Step5 在 cslave3 上将两次计算结果合并起来。

由于单机已无法处理 file9, 只能采取“部分计算→删除→再计算→合并计算”这种新颖办法, 显然该过程要多次删除数据、拷贝数据, 降低了处理速度, 不过若能将这个过程中组织到分布式环境下, 则无须拷贝、删除操作, 且计算过程可以并行执行, 将这种思路重新组织包装, 就是 M-S-R 并行化思想。

问题⑥计算可靠性:

对于问题⑥, 最直观的思路就是为每台机器做磁盘冗余阵列 (RAID), 购买更稳定的硬件, 配置最好的机房、最稳定的网络。

硬件要提供极高的稳定性, 这点没错, 但我们不能“千方百计”地依赖于硬件的可靠性, 最好能在存储和计算这两端都做些冗余, 从软件层预防和处理硬件失败。

6.1.3 分布式解决方案

上述方案并没有真正解决问题, 下面介绍的分布式方案其实就是 MapReduce 处理思路, 读者须仔细研读, 重点理解其处理思想, 至于有些不好理解的地方, 暂不必追究, 后面章节将深入讲解。

1. 分布式存储

本部分内容已经在第 3 章中讲述, 下面直接使用结论, 假定 file0~2 这三个文件在 HDFS 中存储状态如图 6-2 所示, 后面的 M-S-R 并行化计算需要 HDFS 的支持。

2. 分布式计算

针对第二类计算问题, 不可能每次都将数据复制到同一台机器计算, Google 论文 MapReduce 给出观点“移动计算比移动数据更划算”, 试想, 数据动辄就是几个 TB, 而代码一般才几 MB, 如果每次都将程序分发至存储数据的机器上执行, 而不是移动数据,

则处理速度将大大提高。下面采用分布式计算思想解决问题③，先给出如下思路：

首先引入 Key-Value 对概念，称 “<Key,Value>” 为 Key-Value 对，或者 KV 对，大量半结构化数据都可以表示成这种形式，其中 Value 为此 Key 对应的值，且 Key，Value 都可以是复合类型。

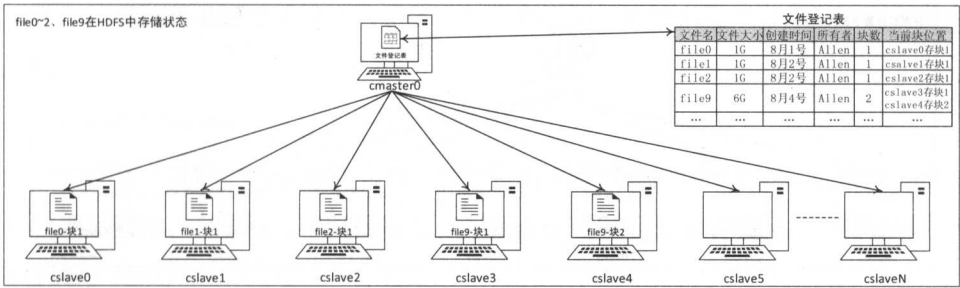


图 6-2 HDFS 中文件存储状态

假定当前 HDFS 块大小为 3G，则这三个文件都只占 1 个块，再假定这三个文件当前存储状态（即 cslaveX 存储 fileX-b1）如图 6-2 所示，file0、file1、file2 内容分别为 “china cstor china”，“cstor china cstor” 和 “china cstor”。首先，在 cslave0、cslave1、cslave2 上，针对各自存储的文件分别独立执行 WordCount 程序（单词计数），结果记成 “<Key,Value>” 形式，其中 Key 为单词，Value 为此 Key 出现次数，如 <cstor,2> 表示单词 cstor 出现 2 次。接着，规定 Key=china 的 <Key,Value> 对前往 cslave5 进行合并计算，Key=cstor 的 <Key,Value> 对前往 cslave6 进行合并计算。然后，cslave5 和 cslave6 分别独立计算汇总至本机的中间结果，并得出最终结果。最后，这两台机器分别将各自计算的最终结果（依旧为 <Key,Value> 形式）存入 DFS（图 6-3）。

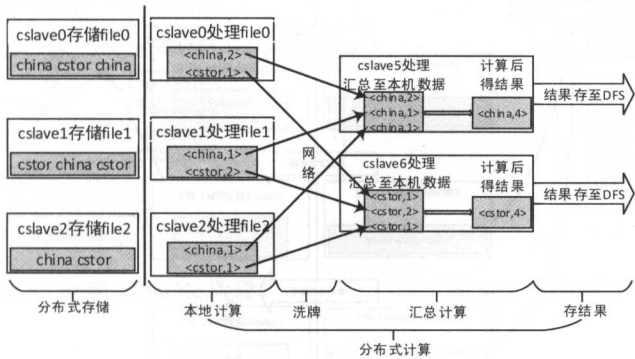


图 6-3 WordCount 分布式计算思路

根据图 6-3，处理过程可大致划分为三步：本地计算（Map）、洗牌（Shuffle）和合并再计算（Reduce），三个过程构建如下：

取新机器 cmaster1, 采用 master/slave 架构构建由机器 cmaster1 和 cslave0~N 组成的分布式计算集群。规定 cmaster1 为 compute master, cslave0~N 为 compute slave, compute master 不处理数据, 主要负责控制 M-S-R 整个处理流程, compute slave 负责处理实际数据, 并且它们还要不断向 compute master 汇报计算进度 (图 6-4)。

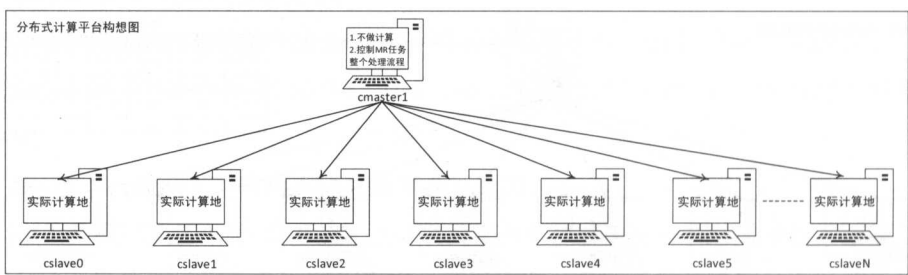


图 6-4 分布式计算架构思想

1) 本地处理 (Map 阶段)

按照“移动计算比移动数据更划算”的思想, cslave0 最好是处理存于本机硬盘上的 file0-b1, 而不是将 file1-b1 从 cslave1 调过来后 (通过网络) 再处理 file1-b1; 同样 cslave1 处理存在 cslave1 上的 file1-b1, cslave2 处理存储在 cslave3 上的 file3-b1, 这就是所谓的“本地计算” (图 6-5), Hadoop 称此过程为分布式计算的 Map 阶段。

由于 file0-b1 存于 cslave0, file1-b1 存于 cslave1, file2-b1 存于 cslave2; 此时当 cslave0 从硬盘上将 file0-b1 读入内存, cslave1 将 file1-b1 读入内存, cslave2 将 file3-b1 读入内存时, 各机读数据的过程是并行的。

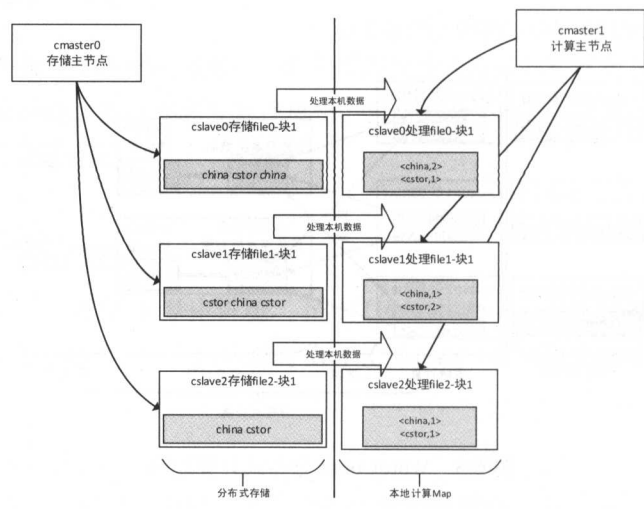


图 6-5 本地计算 (Map 阶段)

上述本地计算 Map 阶段结束后, cslave0~2 分别有中间结果<cs tor,2><china,1>、<cs tor,1><china,2>和<cs tor,1><china,1>。至此,虽已成功实现了本地计算,但只是中间结果,单词计数问题依旧没有解决,容易看出,将这两个中间结果合并,即为最终结果,于是我们自然想到将 cslave1 和 cslave2 上的中间结果复制到 cslave0,在 cslave0 上进行合并计算,可是这种处理方式又变回了单机,那我们如何能够实现“合并”过程也由多机执行呢?

2) 洗牌 (Shuffle 阶段)

为此引入“洗牌”(Shuffle)过程,即规定将 Key 值相同的 KV 对,通过网络发往同一台机器。易知,只要规则相同,那么同一类 Key 所对应的 KV 对们(不管它们原来在哪台机器)经规则后必发往同一台机器。

经过洗牌后, cslave5 上有<china,2>、<china,1>和<china,1>, cslave6 上有<cs tor,1>、<cs tor,2>和<cs tor,1>, 下面进行合并计算(图 6-6)。

3) 再计算 (Reduce 阶段)

合并计算过程稍微微妙些,第一步每台机器将各自 KV 对中的 Value 连接成一个链表,如<china,2><china,1>经连接后成为<china,[2,1,1]>,而 cslave1 则成为<cs tor,[1,2,1]>,接着各台机器可对<Key,ValueList>进行业务处理(如相加),称此过程为 Reduce(图 6-7)。

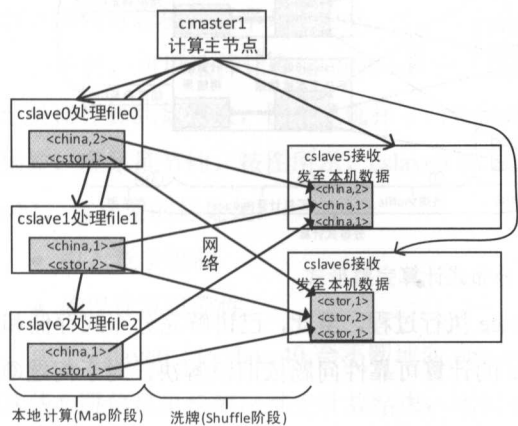


图 6-6 洗牌 (Shuffle 阶段)

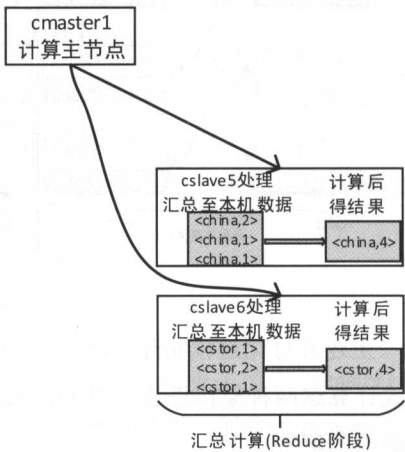


图 6-7 再计算 (Reduce 阶段)

4) 存结果

最后,将得出的结果再存于 DFS,这里若每个 Reduce 得出的结果都存成一个单独文件,则存储结果文件的过程是并行的(图 6-8)。

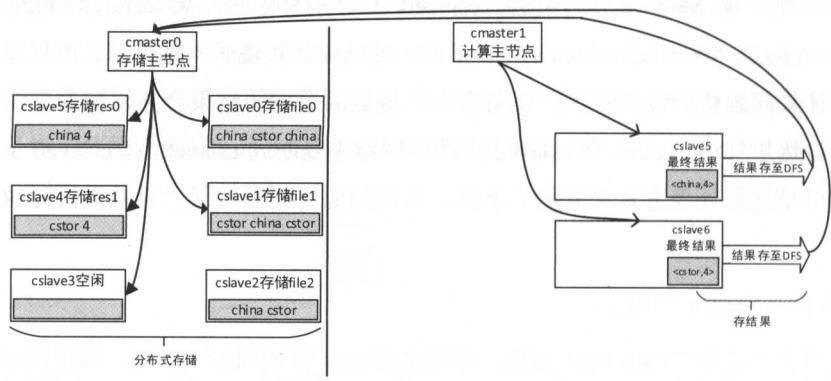


图 6-8 存结果

容易看出，无论是 Map、Shuffle 还是 Reduce，甚至是存储结果，在每个阶段都是并行的，整个过程则构成一个有向无环图（DAG），称此 MapShuffleReduce 过程为 M-S-R 并行化过程或 MapReduce 分布式计算过程，有时也直接称为 MapReduce（图 6-9）。

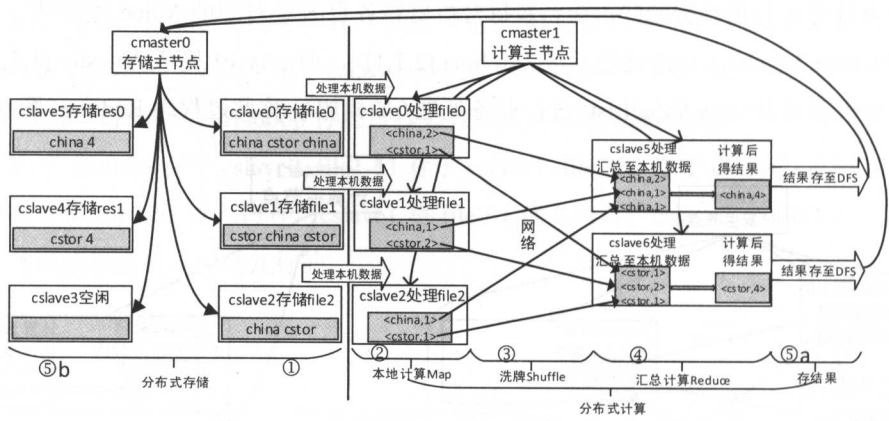


图 6-9 分布式计算完整形式

上述并行化过程实际上就是 MapReduce 执行过程，至此，已讲解完分布式存储和分布式计算这两种架构思路，但 MapReduce 的计算可靠性问题依旧没解决，对于问题⑥，下面给出解答。

3. 冗余计算

由第 4 章可知，HDFS 本身用冗余机制来解决存储可靠性，在 MapReduce 处理过程中，恰巧可以利用这一特性，对同一个数据的不同块，都进行处理（图 6-10）。

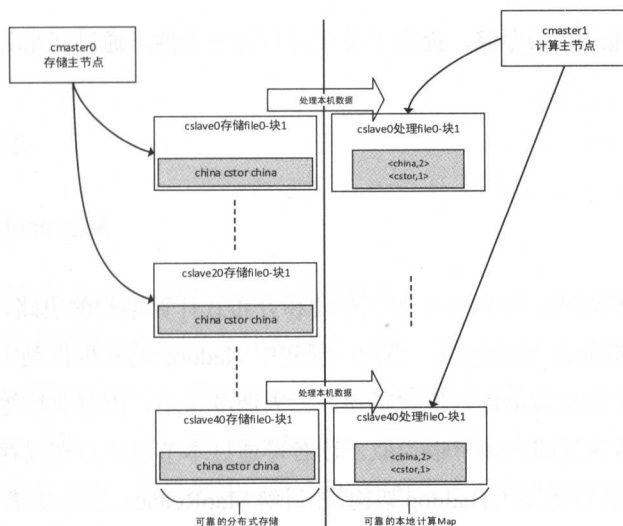


图 6-10 可靠存储和可靠计算

假设存于 `cslave0` 上的 `file0-块1` 还存储于 `cslave20` 和 `cslave40`，由于默认情况下（可设置），HDFS 中相同的数据都有三个数据块，而这三个数据块存储时应当是跨机器，机架甚至是集群的，故，即使单机宕机甚至是硬盘损坏，都不会影响到数据可靠性。假设集群的备份因子设置为 3，HDFS 主服务 `NamNode` 一旦感知到某个数据块数量少于 3 时，就会对此块进行复制操作。假设存储 `file0-块1` 的三台机器属于不同机架、机房，那么这三台机器同时宕机的概率几乎为 0，可以说 HDFS 能够保证 99.9% 的可靠性。

计算时，可以选择让存储 `file0-b1` 的三台机器都同时计算此块数据，也可以只选择两台机器计算此数据块，图中只选择了两台机器，这是因为计算时，CPU 和内存资源相对稀缺，应尽量节约。按图所示，`cslave0` 和 `cslave40` 在同时计算存于本机的相同数据块 `file-b1`，这样做的好处是：

- 引入竞争机制
- 确保计算可靠性

由于 `cslave0` 与 `cslave40` 会不断地向 `cmaster1` 上传心跳包，计算主节点 `cmaster1` 会在这些心跳信息里检测出谁先计算结束，比如 `cslave40` 先计算结束，则此时 `cmaster1` 会向 `cslave0` 发出 Kill 命令，直接让 `cslave0` 停止计算。通过这种竞争，可以尽量确保计算任务尽快完成（图 6-10）。

另一种情况是，当前有两台机器（实际上是进程）在同时计算相同数据块，当 `cslave40` 由于某种原因宕机后，对 `cslave0` 上的计算进程无任何影响，也就是说，对于 `cmaster1` 和用户来说，`file-b1` 的计算并未因此而失败，通过冗余计算的方式，MapReduce 计算也是可靠的。

综上所述,通过冗余存储,提高了分布式存储可靠性,通过冗余计算,提高了分布式计算的可靠性。

6.1.4 小结

本节通过问答方式,引出了分布式存储和分布式计算架构的思路,这也是 Hadoop 的 HDFS 和 MapReduce 架构思路。当然,现实中 Hadoop 的实现机制则更加复杂,比如它的存储与计算都以块为单位、机架感知、三大调度策略、推测执行等,都有其精妙之处,但其架构的基本思路和本节很类似,读者可通过本节理解分布式存储和分布式计算的架构思想,下面章节将深入 Hadoop 架构,在讲解 MapReduce 之前,编者简单叙述 HDFS、YARN 和 MapReduce 之间的关系。

严格地说,这三者之间毫无关系,打个比方,以个人 PC 为例,HDFS 可以看成硬盘(存储介质),YARN 可以看成 Win7(操作系统),MapReduce 可以看成 QQ(应用程序)。Win7 不一定非要安装到硬盘上,QQ 也可以安装到 MacOS、CentOS(WebQQ)下。只不过,Win7 和 QQ 都需要存储介质,而当这种存储介质是硬盘时,性能和成本最佳;QQ 需要操作系统支持才能专注于自身功能,省去关心硬件细节,且当这种操作系统是 Win7 时,市场广阔,经济效益最佳;只有硬盘和 Win7 的机器就像是一个开了机却并未运行任何程序的个人 PC,最多只是“花瓶”,当运行应用程序时,这个 PC 才起作用,实用效率最佳。

同理,HDFS 提供最基础的存储服务,YARN 的功能类似于操作系统,MapReduce 则更像一个应用程序,当这三者在千万台机器(集群)上结合时,效果最佳。

6.2 MapReduce 简介

正如上文所示,以个人 PC 为例,HDFS 可以看成硬盘(存储介质),YARN 可以看成 Win7(操作系统),MapReduce 可以看成 QQ(应用程序)。故在理解 MapReduce 时,可以从 MapReduce 的概念、部署、和 YARN 之间关系,这几个方面入手。又由于集群特殊性,MapReduce 程序是分散在集群中各台机器上执行的,读者还需要知道 MapReduce 的执行拓扑,下面即从这几个方面讲述 MapReduce 框架^[1]。

6.2.1 基本概念

1. MapReduce 由来

2002 年开源组织 Apache 成立开源搜索引擎项目 Nutch, 但在 Nutch 开发过程中, 始终无法有效地将计算任务分配到多台计算机上。2004 年前后, Google 陆续发表三大论文 GFS、MapReduce 和 BigTable。于是 Apache 在其 Nutch 里借鉴了 GFS 和 MapReduce 思想, 实现了 Nutch 版的 NDFS 和 MapReduce。但 Nutch 项目侧重搜索, 而 NDFS 和 MapReduce 则更像是分布式基础架构, 故 2006 年开发人员将 NDFS 和 MapReduce 移出 Nutch, 形成独立项目, 称为 Hadoop。

由于技术和实践的发展, Hadoop 本身也在发展并完善着, 工业界称 Hadoop 1.X 及其以前的版本 (0.23.X 除外) 为 Hadoop 1.0, 称 Hadoop 2.X 及其以后版本为 Hadoop 2.0, 两个版本在 Hadoop 架构上有较大改变。但无论是 Hadoop1.0 还是 Hadoop2.0 中, 都包含 MapReduce 模块, 可以说 MapReduce 理论起源于谷歌, 代码实现于 Hadoop。

2. MapReduce 进程概念

MapReduce 应用程序执行过程中, 主要涉及如下 3 大类 8 个小概念, 其中存储层 HDFS 和操作系统层 YARN 已经在第四、五章讲述, 编者重点讲述应用程序层。

1) 存储层 HDFS

- NameNode
- DataNode
- HDFSClient

HDFS 本身就采用 master/slave 架构, 进程 NameNode 充当主服务, 部署在主节点上; 进程 DataNode 充当从服务, 部署在从节点上。MapReduce 应用程序需要使用 HDFS 的存储功能, 在 HDFS 上存放 MapReduce 的输入数据、输出数据, Jar 文件, 资源数据等。实际上, 当 MapReduce 使用 HDFS 存储数据时, 其就相当 HDFSClient, 访问 HDFS 非常简单, 只需如下一行代码即可:

```
FileSystem hdfs=FileSystem.getFileSystem(Configuration conf);
```

2) 操作系统层 YARN

- ResourceManager
- NodeManager

YARN 本身也采用 master/slave 架构，进程 ResourceManager 充当主服务，部署在主节点上；进程 NodeManager 充当从服务，部署在从节点上。MapReduce 应用程序需要使用 YARN 的编程接口，具体来说，MRClient 向 YARN 提交程序，YARN 会启动本次任务的 MRAppMaster，而 MRAppMaster 在执行过程中需要不断地向 YARN 申请或归还资源（以 Container 形式）。其实这和单机下操作系统上执行应用程序一样，比如 Win7 上的 QQ 在执行过程中也不断地向 Win7 申请或归还内存和 CPU 资源。

3) 应用程序层

- MRClient
- MRAppMaster
- YarnChild
 - Mapper 类
 - Reduce 类

编者在 YARN 编程一节已经讲述，不论是 M 范式、M-S-P 范式、BSP 范式亦或是服务型应用程序，都需要包含 ApplicationBusinessLogic、ApplicationClient 和 ApplicationMaster 这三个模块。作为 M-S-P 范式的代码实现，MapReduce 框架也包含这三个部分，其中 MRClient 对应 MapReduce 框架中的 YarnRunner.java，在机器上实际运行时进程名为 RunJar，编者为了方便理解才使用了这个名字；ApplicationMaster 对应 MRAppMaster，机器上实际运行时进程名就为 MRAppMaster；用户自定义的 Mapper 和 Reduce 类在执行时统称为 YarnChild，表 6-1 即为 MapReduce 框架三大模块。

表 6-1 MapReduce 框架三大模块

YARN 应用程序标准模块	MapReduce 框架对应类	进程名
ApplicationBusinessLogic	用户自定义 Mapper 类、Partition 类、和 Reduce 类	YarnChild
ApplicationClient	YARNRunner.java	RunJar
ApplicationMaster	MRAPPMaster.java	MRAPPMaster

简单地说，MRClient 功能是向 YARN 提交应用程序，MRAppMaster 负责控制整个程序执行流，YarnChild 具体执行业务逻辑层面上的 Mapper 和 Reduce 类。

3. 其他概念

1) YARN

YARN 框架。

2) Yarn-App

Yarn 应用程序，比如 Spark、MapReduce 都是 Yarn-App。

3) MapReduce

MapReduce 框架，其实质上就是 M-S-R 范式的代码实现。

4) MR-App

MapReduce 应用程序，标准的 Yarn-App 包含三部分，MapReduce 框架中只有 MRAppMaster 和 MRClient，加上用户自己编写的 MRAppBusinessLogic (Mapper 类和 Reduce 类)，统称 MR-App。

6.2.2 编程模型

1. 编程模型

MapReduce 采用“分而治之”的思想，把对大规模数据集的操作，分发给一个主节点管理下的各分节点共同完成，然后通过整合各分节点的中间结果，得到最终的结果。简单地说，MapReduce 就是“任务的分解与结果的汇总”。上述处理过程被 MapReduce 高度地抽象为两个函数：map 和 reduce，map 负责处理单个任务，reduce 负责把分解后多任务处理的结果汇总起来。至于在并行编程中的其他种种复杂问题，如分布式存储、作业调度、负载均衡、容错处理、网络通信等，均由 MapReduce 框架负责处理。需要注意的是，用 MapReduce 来处理的数据集必须具备这样的特点：待处理的数据集可以分解成许多小的数据集，而且每一个小数据集都可以完全并行地进行处理。

图 6-11 给出了使用 MapReduce 处理大数据集的过程，从图中可以看出，该计算模型的核心部分是 map 和 reduce 函数。这两个函数的具体功能由用户根据需要自己设计实现，只要能够按照用户自定义的规则，将输入的<key, value>对转换成另一个或一批<key, value>对输出即可。

在 Map 阶段开始之前，JobClient 上交任务之时，JobClient 已经将输入数据在 HDFS 中的分片信息告知了 MRAppMaster，MRAppMaster 会为每一个 split 创建一个 Map 任务（以下简称 Mapper）用于执行用户自定义的 map 函数，并将对应 split 中的<K1, V1>对作为输入，得到计算的中间结果<K2, V2>。接着将中间结果按照 K2 进行排序，并将 key 值相同的 value 放在一起形成一个新列表，形成<K2, list(V2)>元组。最后再根据 key 值的范围将这些元组进行分组，对应不同的 Reduce 任务（以下简称 Reduce）。将这一系列划分好后，Mapper 进程会调用 RPC 模块，将数据发往不同 Reduce 进程。

在 Reduce 阶段，Reduce 把从不同 Mapper 接收来的数据整合在一起并进行排序，然后调用用户自定义的 reduce 函数，对输入的<K2, list(V2)>对进行相应的处理，得到键值对<K3, V3>并输出到 HDFS 上。既然 MapReduce 框架为每个 split 创建一个 Mapper，那

么谁来确定 Reduce 的数目呢？答案是用户。mapred-site.xml 配置文件中有一个表示 Reduce 数目的属性 `mapred.reduce.tasks`，该属性的默认值为 1，开发人员可以通过 `job.setNumReduceTasks()` 方法重新设置该值。

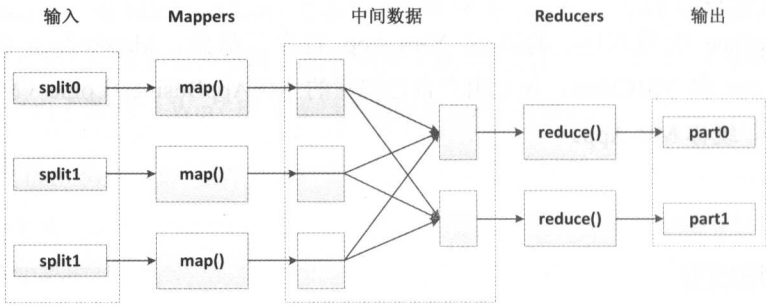


图 6-11 MapReduce 处理大数据集的过程

2. MapReduce 的集群行为

MapReduce 运行在大规模集群之上，要完成一个并行计算，还需要任务调度、本地计算、Shuffle（洗牌）过程等一系列环节共同支撑计算的过程，暂且称之为 MapReduce 的集群行为。

1) 任务调度与执行

MapReduce 任务由一个主进程 MRAppMaster 和多个从进程 YarnChild 共同协作完成。MRAppMaster 主要负责流程控制（即控制 Map→Shuffle→Reduce 整个过程），它通常情况下运行在 NodeManager 的某个 Container 上。MRAppMaster 将 Mapper 和 Reduce 分配给空闲的 Container 后，由 NodeManager 负责启动和监管这些 Container。MRAppMaster 还负责监控任务的运行状况，如果某个 Container 发生故障，MRAppMaster 就会将其负责的任务分配给其他空闲的 Container 重新执行，MRAppMaster 的这种设计很适合于集群上任务的调度和执行。

2) 本地计算

把计算节点和数据节点置于同一台计算机上，MapReduce 框架尽最大的努力保证在那些存储了数据的节点上执行计算任务。这种方式有效地减少了数据在网络中的传输，降低了任务对网络带宽的需求，避免了使网络带宽成为瓶颈，所以“本地计算”可以说是节约带宽最有效的方式，业界称之为“移动计算比移动数据更经济”。也正是因为如此，split 通常情况下应该小于或等于 HDFS 数据块的大小（默认情况下 128MB），从而保证 split 不会跨越两台物理机，便于“本地计算”。

3) Shuffle 过程

MapReduce 会将 Mapper 的输出结果按照 key 值分成 R 份 (R 是预先定义的 Reduces 的个数), 划分时常使用哈希函数, 如 “Hash(key) mod R”。称将划分好的数据发往不同 Reduce 为 Shuffle (洗牌)。

4) 合并 Mapper 输出

正如之前所说, 带宽资源非常宝贵, 所以 MapReduce 允许在 Shuffle 之前先对结果进行合并 (Combine 过程), 即将中间结果中有相同 key 值的一组 <key, valueList> 对合并成一对。其实 Combine 函数就是 Reduce 函数, 但 Combine 过程是 Mapper 的一部分, 在 map 函数之后执行。Combine 过程通常情况下可以有效地减少中间结果的数量, 从而减少数据传输过程中的网络流量。值得注意的是, Hadoop 并不保证其会对一个 Mapper 输出执行多少次 Combine 过程, 也就是说, 开发人员必须保证不论 Combine 过程执行多少次, 得到的结果都是一样的, 根据编者经验, 建议不用 Combine, 稍有不慎很容易出错。

5) 读取中间结果

在完成 Combine 和 Shuffle 的过程后, Mapper 的输出结果被直接写到本地磁盘。然后, 通知 MRAppMaster 中间结果文件的位置, 再由 MRAppMaster 告知 Reduce 到哪个 Mapper 上去取中间结果。注意所有的 Mapper 产生的中间结果均按其 key 值用同一个哈希函数划分成 R 份, R 个 Reduce 各自负责一段 key 值区间。每个 Reduce 需要向多个 Mapper 节点取得落在其负责的 key 值区间内的中间结果, 然后执行 reduce 函数, 形成一个最终的结果文件。需要说明的是, Mapper 的输出结果被直接写到本地磁盘而非 HDFS, 因为 Mapper 输出的是中间数据, 当任务完成之后就可以直接删除了, Spark 此处原理也是一样的, 只不过其中间数据直接放置在内存内, 未写磁盘。

6) 任务管道

R 个 Reduce 会产生 R 个结果, 很多情况下这 R 个结果并不是所需要的最终结果, 而是会将这 R 个结果作为另一个计算任务的输入, 并开始另一个 MapReduce 任务。实际上, 在执行上层的 Hive、Pig 脚本时, 翻译器 (Hive 和 Pig 的翻译器) 会将脚本翻译成多个 MapReduce 和 HDFS 操作, 此时就是所谓的管道机制。不过 Hive 和 Pig 自身封装较好, 若用户需要自己编写 MapReduce 工作流, 可使用 Oozie、Tez 或 Shell。

6.2.3 集群部署

站在用户自己编写的 ApplicationBusinessLogic (Mapper 类、Reduce 类) 角度上,

调度这两个类执行的 MRAppMaster 进程就相当于框架。站在 YARN 角度上, MRAppMaster 只不过是诸多运行在 YARN 上的一个普通应用。上一章已经讲述, 对于 YARN 来说, 所有的 Yarn-App 都是其客户端, 无“安装”这一说法。同理, 无须在 YARN 上安装 MapReduce 框架, 需要执行 MR-App 的客户只要持有 MRClient、MRAppMaster 和 MRAppBusinessLogic (用户自定义 Mapper 和 Reduce 类) 代码, 并使用 MRClient 向 YARN 提交应用程序即可 (图 6-12)。

图 6-12 即为 MapReduce 部署图, 由图中可以看出, 在整个 MapReduce 执行过程中, 只需要提交任务的客户机持有 MRClient、MRAppMaster 和 MRAppBusinessLogic 的代码即可, 无须预先安装 MapReduce 框架, 当 MRClient 成功向 RM 提交本 MR-App 后, RM 会自动调度执行 MRAppMaster, 接着 MRAppMaster 再调度 Container 执行 MRApplicationBusinessLogic (此处为 Mapper 和 Reduce 类)。

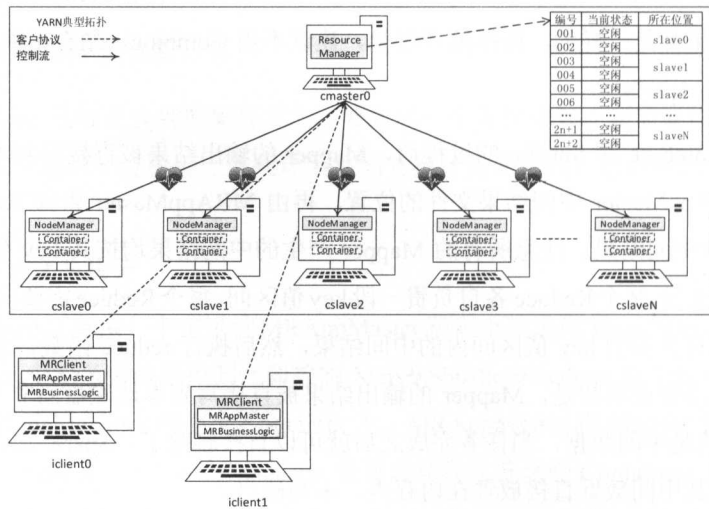


图 6-12 MapReduce 框架部署图

下述命令为向 YARN 提交 MR-App, 奇怪的是该命令只写明了 AppBusinessLogic (wordcount 类里有 Mapper 和 Reduce 两个类), 而并未指明 MRAppMaster 和 MRClient, 这是为何呢?

```
[allen@iclient0 ~]$ yarn jar \
> /usr/hdp/2.2.6.0-2800/hadoop-mapreduce/hadoop-mapreduce-examples-2.6.0.2.2.6.0-2800.jar \
> wordcount /user/allen/in /user/allen/out
```

之所以可以用这个 Shell 命令提交 MR-App, 是因为“yarn”命令本身已经包含了一堆环境变量 (CLASSPATH), 而这些环境变量中一层层引用到了 MRAppMaster.class (MRAppMaster) 和 RunJar.class (MRClient), 故无需在命令行中指明 MRAppMaster 和 MRClient。

6.2.4 体系架构

1. 应用程序层

作为 M-S-R 范式的代码实现，MapReduce 框架应至少实现如下两个功能模块：

- 指挥从 “Data Input→Map→Shuffle→Reduce→Store Result” 整个流程
- Mapper 类和 Reduce 类实际执行体

指挥这个过程的就是进程 MRAppMaster；Mapper 和 Reduce 类的实际执行体则为进程 YarnChild。图 6-13 即为略去 HDFS 和 YARN 的 MapReduce 执行状态图，从图中可以看出，简化后的 MR-App 在运行过程中共包含 client0、MRAppMaster、YarnChild 这三个实体。

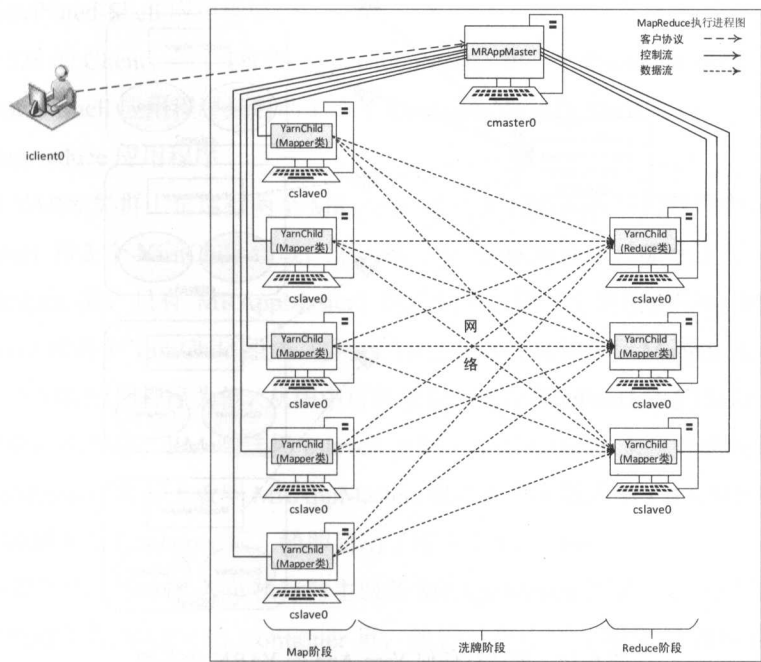


图 6-13 略去 HDFS 和 YARN 的 MapReduce 执行进程图

在 iclient0 的客户代码里有一个 MRAppMaster 的引用，iclient0 可以通过该引用实时查看到本次 MR-App 执行进度。此外，在 MRAppMaster 启动过程中，它会自动启动一个 Web 服务器，并将此 Web UI 暴露给用户。

MRAppMaster 最重要的任务就是控制从 “Data Input→Map→Shuffle→Reduce→Store Result” 到最后的整个流程。而实际执行 Mapper 和 Reduce 类的进程都为 YarnChild，只

不过, 有的 YarnChild 执行 Mapper 代码, 有的执行 Reduce 代码。习惯上称第一层 Container (YARN 术语) 为 Mapper 层, 第二层 Container 为 Reduce 层。

2. 集群资源管理层

MapReduce 的执行需要 HDFS 和 YARN 的支持, 其中 HDFS 主要提供存储支持, YARN 主要提供资源支持。

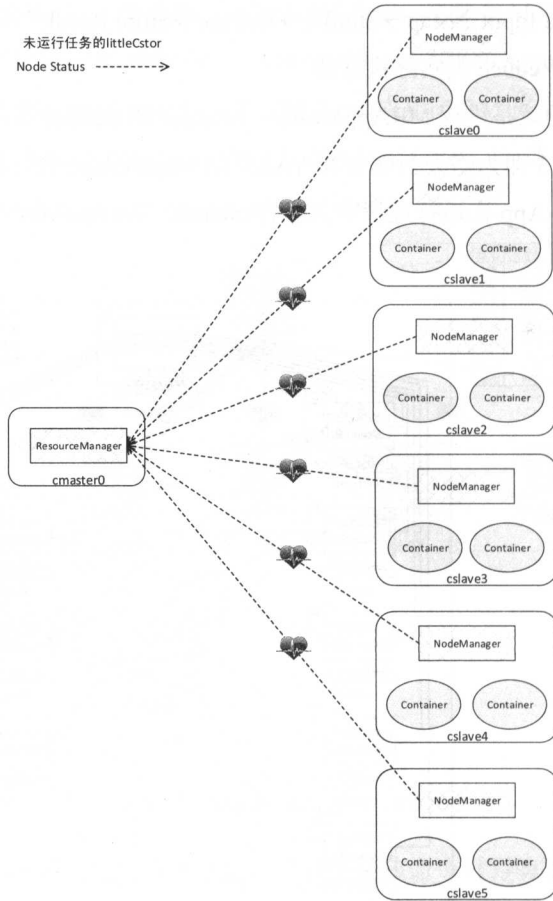


图 6-14 未运行任何 Yarn-App 的 YARN 状态图

由于 HDFS 和 YARN 已分别在第 4、5 章讲述, 故此处只给出 YARN 状态图 (图 6-14), 待会需要用到该图。

3. 工作机制

以单机为例, 可以将 YARN 理解为 Win7, MR-App 理解为 QQ, 就像 PC 上正在运行的 QQ 需要不断向 Win7 申请内存、CPU 等资源一样, YARN 上的 MR-App 在执行过

程中也不断向 YARN 提出资源申请（以 Container 形式）。而当 QQ 运行结束时，Win7 会回收其资源，同理，当 MR-App 执行结束时，YARN 也会回收其资源。

未运行任何应用程序的 YARN 集群中只存在 ResourceManager 和 NodeManager 两大实体（图 6-14），其中 ResourceManager 为集群资源管理者，对内负责管理资源，对外负责作业调度；NodeManager 为单机资源管理者，主要负责管理本机资源（以 Container 形式），NodeManager 会定期向 ResourceManager 上传心跳包，以证明自身存活并汇报 Container 状态。

假定有一个正在运行三个 Yarn-App 的 YARN 集群（图 6-15），以下即以该集群为例，讲述任务提交和执行过程。正如图中所示，在这三个应用中，有一个 Distributed-Shell，两个 MR-App。这个状态图看似复杂实际内容并不多，其主要包括如下内容：

（1）YARN

包含运行在 cmaster0 上的 ResourceManager 和运行在 cslaveX 上的 NodeManager。

（2）Distributed-Shell 应用程序

由一个 DS 的 Client、一个 DS 的 ApplicationMaster 和一个 Container 组成，也就是说，这个 Distributed-Shell 应用程序只启动了一个 Container 来执行 Shell 命令。

（3）MapReduce 应用程序

此时的 YARN 集群上正运行两个 MR-App，第一个 MR-App 应用程序由 MRClient1、MRAppMaster1 和 3 个 YarnChild 组成，至于这三个 YarnChild 中，哪个运行 Mapper 类，哪个执行 Reduce 类，只有 MRAppMaster1 自己知道。第二个应用程序由 MRClient2、MRAppMaster2 和两个 YarnChild 组成，这两个 YarnChild 会具体执行 Mapper 或 Reduce 类。

以第一个 MR 应用程序为例，从图中可以看出，运行在 iclient1 上的 MRClient1 进程会向 RM 提交应用程序，RM 则选择 cslave3 上的一个 Container 来执行此应用程序的主服务 MRAppMaster1（实际上就叫 MRAppMaster，编者为了区别才加了 1），MRAppMaster1 在向 RM 申请到 3 个 Container 后，随即启动了这 3 个 Container。

图 6-16 即为第一个 MR-App 执行时主服务 MRAppMaster 及其三个从服务 YarnChild 状态图，它们宿主在 YARN 的 Container 里，执行过程中会不断向 YARN 申请或归还资源。

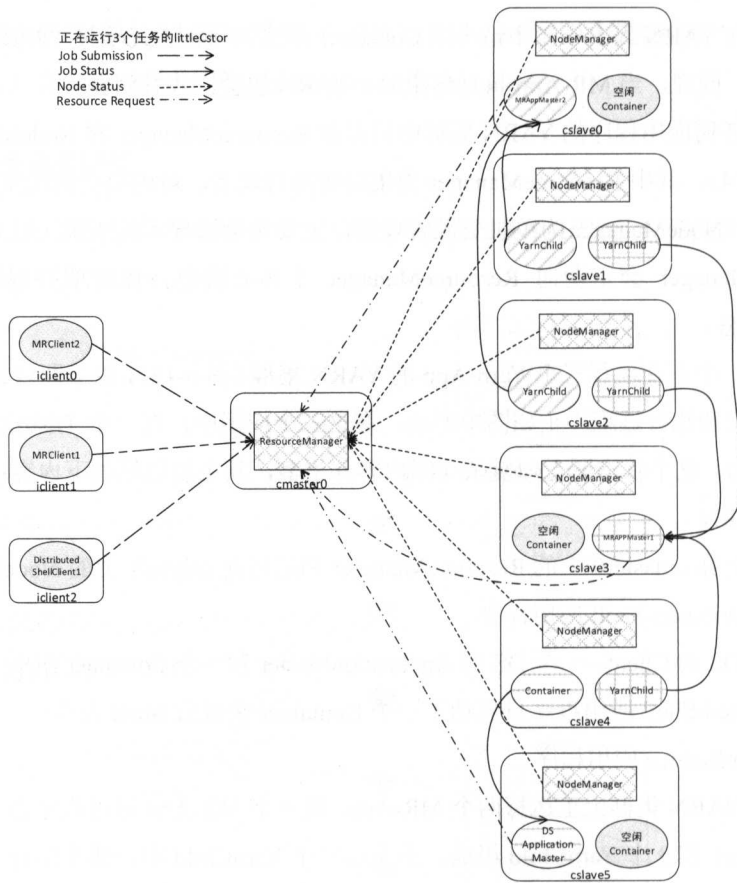


图 6-15 正在运行三个 Yarn-App 的 YARN 状态图

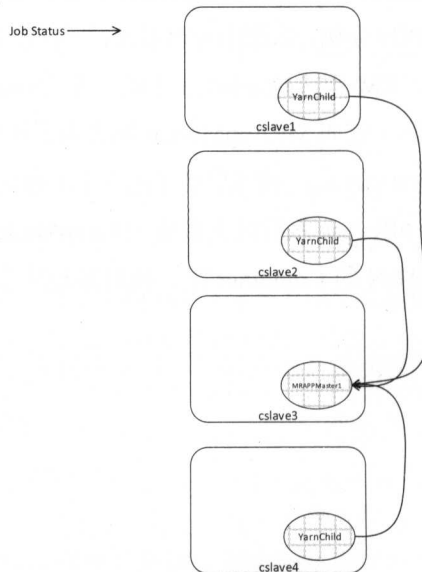


图 6-16 第一个 MR 应用程序执行状态图

6.2.5 执行过程

在 MR-App 在执行过程中，主要包含“Data Input→Map→Shuffle→Reduce→Store Result”这几个阶段，无论是控制这个过程的 MRAppMaster，还是执行 Mapper 与 Reduce 类的 YarnChild，都需要 YARN 交互，下面即讲述 YARN 中的 MR-App 的执行流程，在讲解之前，先罗列本节涉及的各大实体。

- MRClient: 提交 MapReduce 作业的客户端。
- ResourceManager: YARN 资源管理器，负责资源管理、作业管理、资源仲裁。
- NodeManager: 节点管理器，负责启动和监视本节点上的计算容器 Container。
- MRAppMaster: MapReduce 类应用程序主进程，负责控制整个应用程序执行。
- YarnChild: 执行 MapTask 或 ReduceTask 的进程名。
- HDFS: 分布式文件系统，MR-App 输入文件和 MR-App 代码存放地。

在 YARN 上执行一个 MR-App 时，主要包含如下六个步骤，下面将具体讲述这六个步骤。

- Step1 作业提交（站在客户端角度）。
- Step2 作业初始化（站在 YARN 角度）。
- Step3 任务分配（站在 MRAppMaster 角度）。
- Step4 任务执行（站在 MRAppMaster 角度）。
- Step5 进度和状态更新（站在 MRAppMaster 角度）。
- Step6 作业完成（站在 MRAppMaster 角度）。

1. 作业提交

当 MRClient 向 RM (ResourceManager) 提交 MR-App 后，RM 会对客户端提交的作业信息做出基本验证，一旦通过验证，RM 会将 AppID 返回给 MRClient（图 6-17 中步骤 2），注意此时作业并未真正提交，MRClient 还需要到 HDFS 上检查本应用程序输出目录，计算输入分片，然后将作业资源（包括 MRAppMaster 的 Jar 包、ApplicationBusinessLogic 的 Jar 包，配置，分片信息等）复制到 HDFS 上（图 6-17 中步骤 3）。在准备好 MRAppMaster 的上下文信息并将这些信息写入 appContext 后，客户端 MRClient 须通过调用 yarnClient.submitApplication (appContext) 向集群提交作业（图 6-17 中步骤 4），此时才算是真正提交作业。在作业提交后，客户端进程 MRClient 可根据 AppID 追踪作业执行进度。

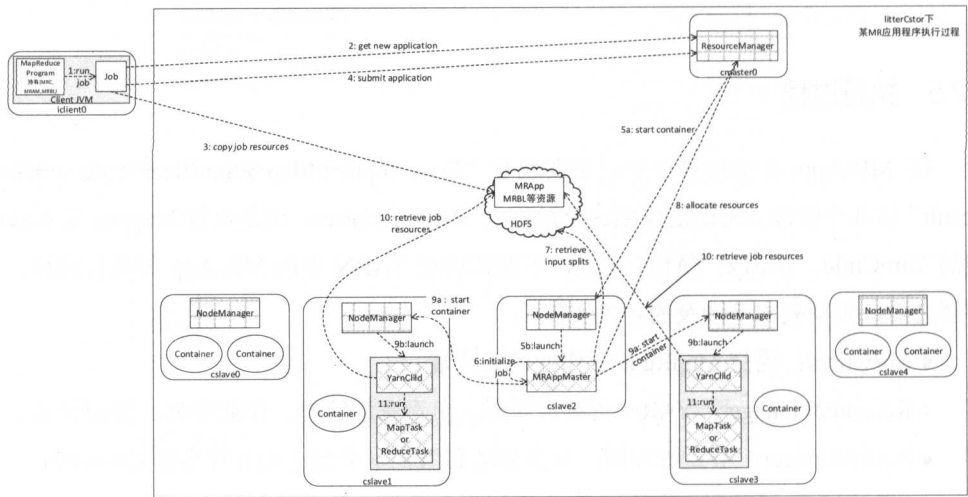


图 6-17 YARN 上运行 MapReduce 任务过程

2. 作业初始化

正常情况下，在 RM 收到一个新的应用程序后，它首先对这个应用程序进行安全检验，一旦通过检验，便将请求传递给 Scheduler。Scheduler 会为该应用程序分配一个 Container 来执行应用程序的主服务 MRAppMaster，正如前文所述，习惯上，称执行 MRAppMaster 的 Container 为 Container0，接着 RM 通过 NM 启动 Container0（步骤 5a 和 5b），Container0 上运行的进程即为 MRAppMaster。

MRAppMaster 会作为本次应用的总控进程（习惯上称为 MR-App 主进程）控制本应用程序整个执行流程。它（进程 MRAppMaster）首先会对作业进行初始化，然后创建相关跟踪对象以跟踪所有 Container 的执行进度（步骤 6），接着，MRAppMaster 进程获取输入数据的分片信息（步骤 7），对每一个分片申请一个 Container（步骤 8），这些 Container 里都将执行 ApplicationBusinessLogic 里的 Mapper 类。按系统指定的 Reduce 个数，再申请相应数量的 Container（步骤 8），这些 Container 里都执行 ApplicationBusinessLogic 里的 Reduce 类。

3. 任务分配

接下来，MRAppMaster 以心跳包方式到 RM 处申请应用程序需要的 Container 数（步骤 8），其中心跳包里会包含本 Container 的偏好信息，MapReduce 程序中这种偏好主要为本地计算，RM 会根据系统资源状况尽量满足 MRAppMaster 的所有申请。

4. 任务执行

在申请到一定数量的 Container 后, MRAppMaster 内部的 Scheduler (和 RM 的 Scheduler 不同) 模块会将 ApplicationBusinessLogic 下沉至 Container 里并通过 NodeManager.startContainerAsync(), 逐个启动这些 Container。在这些 Map 或 Reduce 类型的 Container 执行前, Scheduler 模块还须为此 Container 配置完备的上下文信息, 比如此 Container 关联的资源, Jar 文件, 输入文件, 执行令牌等, 这样, 当 Container 真正执行时, 它能够将这些执行所需资源 (Shell 命令、Jar 包、输入文件、令牌等) 拷至本地, 最后 NM 和本地 OS 交互, 通过执行上下文信息里的 Shell 命令, 执行此 Container (Mapper 或 Reduce 任务) (步骤 11)。

5. 进度和状态更新

在各个 Container 执行时, 这些 Container (进程名为 YarnChild) 会每三秒向 MRAppMaster 汇报进度和状态信息, MRAppMaster 则将这些信息汇聚到一起并据此计算任务执行进度, 存档任务执行过程和给用户提供进度查询。

6. 作业完成

当执行完所有 Container 后, MRAppMaster 会向 RM 汇报任务结束并注销自己, 在向 MRAppMaster 上报的心跳包里, MRAppMaster 会写明已释放的 Container 资源信息, RM 的 Scheduler 模块会异步回收这些计算资源。

在程序运行过程中, MRClient 会每隔 1 秒查询一下 MRAppMaster 执行进度, 在检测到任务成功执行后, MRClient 会结束自己, 至此, 本应用程序涉及的三大计算实体, iclient0 上的 MRClient 进程 (实际上进程名 RunJar)、cslave2 上的 MRAppMaster 进程和 cslave1、3 上的 YarnChild 进程都成功结束。

6.3 MapReduce 接口

MapReduce 接口指的是用户取得 MapReduce 服务的途径, 下面先讲述常见的 MapReduce 接口, 接着直接讲述 Web 接口。

1. 接口汇总

作为大数据处理领域最常用的并行计算框架, 针对不同的上层应用, MapReduce 框

架提供了三类统一访问接口，分别为：

- MapReduce 自带 Web 接口
- MapReduce Shell 接口
- MapReduce Java API 接口

Web 接口主要为管理员提供，从该页面上，管理员能看到已经完成的所有 MR-App 执行过程中的统计信息，该页面只支持读，不支持写操作。

Shell 接口主要针对 MapReduce 程序员，通过 Shell 接口，程序员能够向 YARN 集群提交 MR-App，查看正在运行的 MR-App，甚至可以终止正在运行的 MR-App，6.4 节重点 MapReduce 的 Shell 接口。

MapReduce Java API 面向 Java 开发工程师，程序员可以通过该接口编写 MR-App 用户层代码 MRApplicationBusinessLogic，6.5~6.7 节将讲述 MapReduce 编程。

2. 实战 MapReduce Web

由于 MapReduce Web 接口内容较少，此处直接讲解。MapReduce Web 的默认地址是“jobhistoryIP:19888”，而在 littleCstor 中 historyserver 服务部署在 cmaster1 上，故此处在浏览器中地址栏输入“cmaster1.cloudlab.njupt.edu:19888”即可进入 JobHistory 主界面（图 6-18、图 6-19）。



图 6-18 JobHistory 主界面 1



图 6-19 JobHistory 主界面 2

读者可能会奇怪，既然在 YARN 集群并未运行任何 Yarn-App 时，MapReduce 框架并不存在，又哪来的 MapReduce Web 界面？JobHistory 又是什么？这是因为，为进一步减轻 YARN 负担，MapReduce 开发团队新开发了一个“historyserver”，负责管理所有运

行结束的 Yarn-App。这样，YARN Web 界面只管理正在运行的 Yarn-App，一旦该 Yarn-App 运行结束，直接交予“historyserver”管理。诚然 YARN 自身也可以管理那些已经运行结束的 Yarn-App，可随着时间的推移，YARN 自身会变得越来越吃力。比如，你的 YARN 集群已经运行了一万个 Yarn-App，当你非要让 YARN 来维护这一万个已经完成的 Yarn-App，YARN 将变得非常低效。

为防止 YARN 变得越来越慢，设计者引入了“historyserver”，让其管理已经执行结束的 Yarn-App。

当 Client 向 YARN 提交任务时，直至运行结束之前，该 Yarn-App 都会显示在 YARN Web 里（图 6-20），当任务运行结束后，“ApplicationMaster”链接会变成“History”链接，点击该链接，即可进入“JobHistory”页面（图 6-19）。

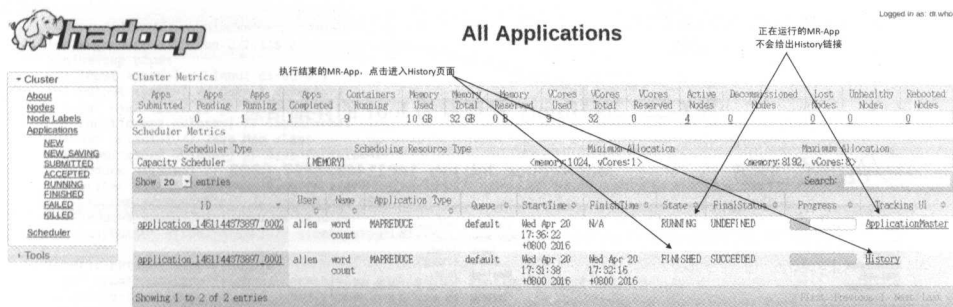


图 6-20 YARN 主界面

当然，读者也可以在浏览器地址栏中输入“cmaster1.cloudlab.njupt.edu:19888”，直接进入“JobHistory”页面，比如当图 20 中的两个 Yarn-App 执行完成后，JobHistory 页面显示如图 6-21 所示。

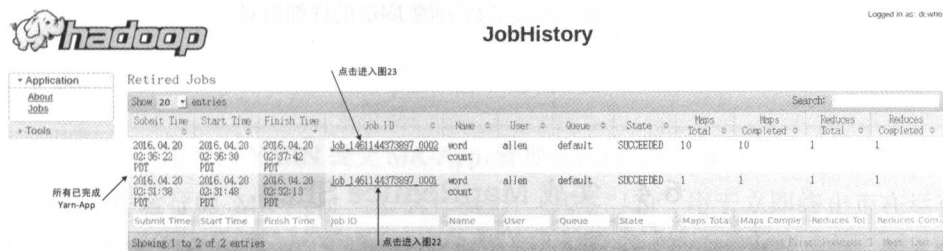


图 6-21 JobHistory 主界面

点击 JobHistory 上的 Job 链接，或点击 YARN Web 上的“History”链接，皆可进入 Job 详情页，该页显示了该 Yarn-App 执行时的所有日志和统计信息。如图 6-22 为 Job1 执行过程中的详细信息，图 6-23 为 Job2 执行过程中的详细信息。

通过 JobHistory 页面，一方面进一步降低了 YARN 负担，另一方面统一管理着已完成的 Yarn-App，让程序员能够无论过多久都可重新查看程序执行过程，大大方便程序调试。

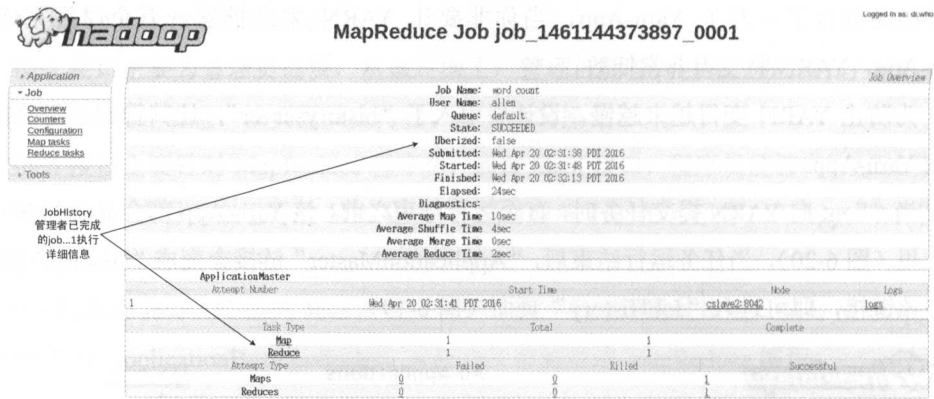


图 6-22 JobHistory 管理着的 Job1 的详细信息

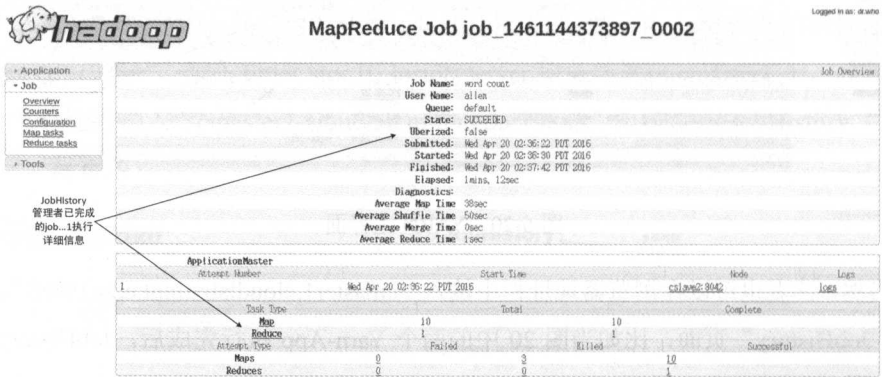


图 6-23 JobHistory 管理着的 Job2 的详细信息

6.4 实战 MapReduce Shell

MapReduce Shell 接口是 MapReduce 功能的实际体现，用户可以使用 MapReduce Shell 命令完成任务提交、查看、管理等工作。

MapReduce Shell 入口统一，其所有命令的第一标签都在“mapred”命令下，图 6-24 为 MapReduce 命令行统一入口。下面列举常用的“mapred”命令，请读者参考编者演示进行练习。

```
[allen@cmaster0 hadoop-2.7.1]$ bin/mapred
Usage: mapred [--config confdir] [--loglevel loglevel] COMMAND
where COMMAND is one of:
  pipes          run a Pipes job
  job            manipulate MapReduce jobs
  queue          get information regarding JobQueues
  classpath      prints the class path needed for running
                 mapreduce subcommands
  historyserver  run job history servers as a standalone daemon
  distcp <srcurl> <desturl> copy file or directories recursively
  archive -archiveName NAME -p <parent path> <src>* <dest> create a hadoop archive
  hsadmin        job history server admin interface

Most commands print help when invoked w/o parameters.
[allen@cmaster0 hadoop-2.7.1]$
```

图 6-24 MapReduce 命令统一入口

1. pipes

该命令用于向 YARN 集群提交 MR-App，实际上该命令功能和“yarn jar”类似（图 6-25）。

```
[allen@cmaster0 hadoop-2.7.1]$ bin/mapred pipes
bin/hadoop pipes
[-input <path>] // Input directory
[-output <path>] // Output directory
[-jar <jar file>] // jar filename
[-inputformat <class>] // InputFormat class
[-map <class>] // Java Map class
[-partitioner <class>] // Java Partitioner
[-reduce <class>] // Java Reduce class
[-writer <class>] // Java RecordWriter
[-program <executable>] // executable URI
[-reduces <num>] // number of reduces
[-lazyOutput <true/false>] // createOutputLazily

Generic options supported are
-conf <configuration file>      specify an application configuration file
-D <property=value>             use value for given property
-fs <local|namenode:port>       specify a namenode
-jt <local|resourcemanager:port> specify a ResourceManager
-files <comma separated list of files> specify comma separated files to be copied to the map reduce cluster
-libjars <comma separated list of jars> specify comma separated jar files to include in the classpath.
-archives <comma separated list of archives> specify comma separated archives to be unarchived on the compute
machines.

The general command line syntax is
bin/hadoop command [genericOptions] [commandOptions]

16/04/17 08:13:24 INFO util.ExitUtil: Exiting with status 1
[allen@cmaster0 hadoop-2.7.1]$ bin/yarn jar
RunJar jarFile [mainClass] args...
[allen@cmaster0 hadoop-2.7.1]$
```

图 6-25 pipes 命令

2. job

该命令主要用来向 YARN 提交 MR-App，管理正在运行的 MR-App。比如参数“status”用于查看正在运行的 MR-App 当前执行状态；参数“kill”用于立即终止正在运行的 MR-App；参数“set-priority”用于设置正在运行的 MR-App 的运行优先级。图 6-26、图 6-27 演示了“job”命令常用的几个选项。

需要指出的是，参数“set-priority”是无效的，这是因为该命令只适用于 Hadoop1.0 版作业调度器，而 YARN 的中作业优先级全部交由队列控制，用户可采用多级队列实现实现公有云。

```
[allen@cmaster0 hadoop-2.7.1]$ bin/mapred job
Usage: CLI <command> <args>
  [-submit <job-file>]
  [-status <job-id>]
  [-counter <job-id> <group-name> <counter-name>]
  [-kill <job-id>]
  [-set-priority <job-id> <priority>]. Valid values for priorities are: VERY_HIGH HIGH NORMAL LOW VERY_LOW
  [-events <job-id> <from-event-#> <#-of-events>]
  [-history <jobHistoryFile>]
  [-list [all]]
  [-list-active-trackers]
  [-list-blacklisted-trackers]
  [-list-attempt-ids <job-id> <task-type> <task-state>]. Valid values for <task-type> are MAP REDUCE. Valid values for <task-state> are running, completed
  [-kill-task <task-attempt-id>]
  [-fail-task <task-attempt-id>]
  [-logs <job-id> <task-attempt-id>]

Generic options supported are
-conf <configuration file>      specify an application configuration file
-D <property=value>             use value for given property
-fs <local|namenode:port>       specify a namenode
-jt <local|resourcemanager:port> specify a ResourceManager
-files <comma separated list of files> specify comma separated files to be copied to the map reduce cluster
-libjars <comma separated list of jars> specify comma separated jar files to include in the classpath.
-archives <comma separated list of archives> specify comma separated archives to be unarchived on the compute machines.

The general command line syntax is
bin/hadoop command [genericOptions] [commandOptions]

[allen@cmaster0 hadoop-2.7.1]$ bin/mapred job -list
16/04/17 08:32:01 INFO client.RMPProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
Total jobs:3
ners    UsedMem    RsvdMem    State    StartTime    UserName    Queue    Priority    UsedContainers    RsvdContai
job_1460883477129_0015    RUNNING    1460907163942    allen    default    NORMAL    2
0    3072M    0M    1460907167518    allen    default    NORMAL    2
job_1460883477129_0016    RUNNING    3072M    0M    http://cmaster0:8088/proxy/application_1460883477129_0016/    0
job_1460883477129_0017    PREP    1460907169792    allen    default    NORMAL    0
0    0M    http://cmaster0:8088/proxy/application_1460883477129_0017/

[allen@cmaster0 hadoop-2.7.1]$ bin/mapred job -set-priority job_1460883477129_0017 VERY_HIGH
16/04/17 08:32:12 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform. Using builtin-java classes where applicable
16/04/17 08:32:13 INFO client.RMPProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
Changed job priority.
[allen@cmaster0 hadoop-2.7.1]$
```

job命令统一入口

list: 显示正在运行的Yarn-App

将job...17优先级设为最高级

图 6-26 job 命令下常用选项 1

```
[allen@cmaster0 hadoop-2.7.1]$ bin/mapred job -status job_1460883477129_0018
16/04/17 08:44:35 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform.
Using builtin-java classes where applicable
16/04/17 08:44:35 INFO client.RMPProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032

Job: job_1460883477129_0018
Job File: hdfs://cmaster0:8020/tmp/hadoop-yarn/staging/allen/.staging/job_1460883477129_0018/job.xml
Job Tracking URL : http://cmaster0:8088/proxy/application_1460883477129_0018/
Uber job : false
Number of maps: 10
Number of reduces: 1
map() completion: 0.3
reduce() completion: 0.0
Job state: RUNNING
retired: false
reason for failure:
Counters: 33
  File System Counters
    FILE: Number of bytes read=0
    FILE: Number of bytes written=348527
    FILE: Number of read operations=0
    FILE: Number of large read operations=0
    FILE: Number of write operations=0
    HDFS: Number of bytes read=6554
    HDFS: Number of bytes written=0
    HDFS: Number of read operations=9
    HDFS: Number of large read operations=0
    HDFS: Number of write operations=0
  Job Counters
    Launched map tasks=10
    Launched reduce tasks=1
    Data-local map tasks=10
    Total time spent by all maps in occupied slots (ms)=63602
    Total time spent by all map tasks (ms)=63602
    Total vcore-seconds taken by all map tasks=63602
    Total megabyte-seconds taken by all map tasks=65128448
  Map-Reduce Framework
    Map input records=230
    Map output records=2070
    Map output bytes=14274
    Map output materialized bytes=2315
    Input split bytes=336
    Combine input records=2070
    Combine output records=254
    Spilled Records=254
    Failed Shuffles=0
    Merged Map outputs=0
    GC time elapsed (ms)=1337
    CPU time spent (ms)=2380
    Physical memory (bytes) snapshot=586694656
    Virtual memory (bytes) snapshot=6162026496
    Total committed heap usage (bytes)=363540480
  File Input Format Counters
    Bytes Read=6218

[allen@cmaster0 hadoop-2.7.1]$ bin/mapred job -kill job_1460883477129_0018
16/04/17 08:44:58 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform.
Using builtin-java classes where applicable
16/04/17 08:44:58 INFO client.RMPProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
Killed job job_1460883477129_0018
[allen@cmaster0 hadoop-2.7.1]$
```

查看job18当前执行状态

Map完成 30%

reduce还未开始

虽然job正在运行

直接终止正在运行的job

图 6-27 job 命令下常用选项 2

3. queue

该命令主要用来查看 YARN 队列资源分配量，图 6-28 展示了“queue”命令入口方式，图 6-29 则展示了参数“list”、“info”，从“list”中可以看出，新装的 littleCstor 中只配置了“default”队列，调度策略为“capacity”。

```
[allen@cmaster0 hadoop-2.7.1]$ bin/mapred queue
Usage: JobQueueClient <command> <args>
    [-list]
    [-info <job-queue-name> [-showJobs]]
    [-showacIs]

Generic options supported are
-conf <configuration file>      specify an application configuration file
-D <property=value>             use value for given property
-fs <local|namenode:port>       specify a namenode
-jt <local|resourcemanager:port> specify a ResourceManager
-files <comma separated list of files> specify comma separated files to be copied to the map reduce cluster
-libjars <comma separated list of jars> specify comma separated jar files to include in the classpath.
-archives <comma separated list of archives> specify comma separated archives to be unarchived on the compute machines.

The general command line syntax is
bin/hadoop command [genericOptions] [commandOptions]

[allen@cmaster0 hadoop-2.7.1]$
```

图 6-28 queue 命令统一入口

当采用“capacity”调度策略时，资源分配策略非常弹性，以图 6-29 中正在运行的“job18”和“job19”为例，当只运行“job18”时，队列“default”会将其持有的所有资源全部分配给“job18”；当集群中同时出现“job18”和“job19”时，队列“default”会将其持有的所有资源均匀分配给这两个任务；当“job18”运行一段时间后，“job19”才提交时，队列“default”会将“job18”释放的资源分配给“job19”。总之，“capacity”调度策略弹性非常大，其最大优点是充分利用集群资源。

```
[allen@cmaster0 hadoop-2.7.1]$ bin/mapred queue -list
16/04/17 09:08:50 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes
where applicable
16/04/17 09:08:50 INFO client.RMProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
=====
Queue Name : default
Queue State : running
Scheduling Info : Capacity: 100.0, MaximumCapacity: 100.0, CurrentCapacity: 0.0
[allen@cmaster0 hadoop-2.7.1]$ bin/mapred queue -info default -showJobs
16/04/17 09:09:09 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes
where applicable
16/04/17 09:09:10 INFO client.RMProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
=====
Queue Name : default
Queue State : running
Scheduling Info : Capacity: 100.0, MaximumCapacity: 100.0, CurrentCapacity: 0.0
Total jobs:0
=====
ners UsedMem RsvdMem State StartTime Username Queue Priority UsedContainers RsvdContai
[allen@cmaster0 hadoop-2.7.1]$ bin/mapred queue -list
16/04/17 09:12:20 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes
where applicable
16/04/17 09:12:20 INFO client.RMProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
=====
Queue Name : default
Queue State : running
Scheduling Info : Capacity: 100.0, MaximumCapacity: 100.0, CurrentCapacity: 75.0
[allen@cmaster0 hadoop-2.7.1]$ bin/mapred queue -info default -showJobs
16/04/17 09:12:27 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes
where applicable
16/04/17 09:12:27 INFO client.RMProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
=====
Queue Name : default
Queue State : running
Scheduling Info : Capacity: 100.0, MaximumCapacity: 100.0, CurrentCapacity: 65.625
Total jobs:3
=====
ners UsedMem RsvdMem State StartTime Username Queue Priority UsedContainers RsvdContai
job_1460683477129_0022 PREP 1460999480941 allen default NORMAL 6
job_1460683477129_0026 RUNNING 1460999481741 allen default NORMAL 8
job_1460683477129_0021 RUNNING 1460999485677 allen default NORMAL 11
[allen@cmaster0 hadoop-2.7.1]$ bin/mapred queue -list
16/04/17 09:13:53 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes
where applicable
16/04/17 09:13:53 INFO client.RMProxy: Connecting to ResourceManager at cmaster0/192.168.170.133:8032
=====
Queue Name : default
Queue State : running
Scheduling Info : Capacity: 100.0, MaximumCapacity: 100.0, CurrentCapacity: 0.0
[allen@cmaster0 hadoop-2.7.1]$
```

图 6-29 queue 命令实例

4. 其他 Shell 命令

参数“classpath”用于显示 MapReduce 框架完整环境变量, 该环境变量对于 MapReduce 程序员非常重要。

参数“historyserver”用于手动启动 JobHistory 进程, 前面已经讲述, YARN 会将已经完成的 Yarn-App 交予 JobHistory 管理, 显然, JobHistory 并不是必选项, 因为当 littleCstor 中不安装 JobHistory 时, 大不了不查看已完成 Yarn-App 的历史记录, 其对 YARN 集群不能执行 Yarn-App, 无任何影响。

参数“distcp”性能强大, 实用性并不高, 该命令主要用于实现两个 Hadoop 集群间大规模拷贝数据, 比如现有两个 Hadoop 集群 clusterA 和 clusterB, 当希望将 clusterA 中 100TB 数据快速拷贝至 clusterB 时, 可使用“distcp”, 实质上, 该命令内部使用 MapReduce 程序分布式拷贝数据。

参数“archive”用于归档 HDFS 中文件, 可将其看成 Win7 上、将多个文件压缩成一个文件的压缩工具。

6.5 实战 MapReduce 编程

基于 YARN 编写 MR-App 和基于 Hadoop1.0 编写 MR-App 编程步骤相同, 二者完全兼容, 下面给出 MR-App 编写步骤, 在第 7~9 章给出编程实例。

1. MR-App 编写步骤

MR-App 的编写步骤和普通的 Yarn-App 相同, 包含如下三步:

Step1 编写 MRApplicationBusinessLogic

Step2 编写 MRApplicationMaster

Step3 编写 MRApplicationClient

由于 Hadoop 开发人员已经针对 MRClient 和 ApplicationMaster, 开发了它们的通用实现 YARNRunner.java 和 MRAppMaster.java, 故实际开发 MR-App 时, 程序员只需要编写 MRApplicationBusinessLogic。

实际上, 所谓的 YARN 下的 MapReduce 框架, 指的就是 MRAppMaster.java 和 YARNRunner.java。不过作为一个标准 MR-App, 肯定需要业务模块, 即用户根据业务需求自己编写的 MRApplicationBusinessLogic。

有了高效的 MRAppMaster 和 YARNRunner, 用户编写 MR-App 就简单多了, 只需要

按照 MRApplicationBusinessLogic 编写流程，编写该模块即可。

2. MRApplicationBusinessLogic 编写步骤^[2]

MRApplicationBusinessLogic 模块编写过程非常简单，在实际操作时，最常用的编程步骤如下。

Step1 确定<key,value>对

<key,value>对是 MapReduce 编程框架中基本的数据单元，其中 key 实现了 WritableComparable 接口，value 实现了 Writable 接口，这使得框架可以对其序列化并可以对 key 执行排序。

Step2 定制输入格式

InputFormat、InputSplit、RecordReader 是数据输入的主要编程接口。InputFormat 主要实现的功能是将输入数据分切成多个块，每个块都是 InputSplit 类型；而 RecordReader 负责将每个 InputSplit 块分解成多个<key1,value1>对传递给 Map 任务。

Step3 Mapper 阶段

此阶段涉及的编程接口主要有 Mapper、Reducer、Partitioner。实现 Mapper 接口主要是实现其 Map 方法，Map 主要用来处理输入<key1,value1>对并产生输出<key2,value2>对。在 Map 处理过<key1,value1>对之后，可以实现一个 Combiner 类对 Map 的输出进行初步的规约操作，此类实现了 Reducer 接口。而 Partitioner 主要是根据 Map 的输出<key2,value2>对的值，将其分发给不同 Reduce 任务。

Step4 Reducer 阶段

此阶段需要实现 Reducer 接口，主要是实现 Reduce 方法，框架将 Map 输出的中间结果根据相同的 key2 组合成<key2,list(value2)>对作为 Reduce 方法的输入数据并对其进行处理，同时产生输出数据<key3,value3>对。

Step5 定制输出格式

数据输出阶段主要实现两个编程接口，其中 FileOutputFormat 接口用来将数据输出到文件，RecordWriter 接口负责输出一个<key,value>对。

将上述流程稍加整理，可概括出流程图（图 6-30），MR 程序员只需要按照此流程图，定制业务所需类型即可，在稍后的实战环境将看到，几乎所有的 MRApplicationBusinessLogic 都是这个编写流程。

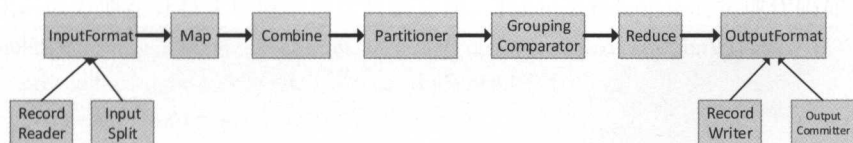


图 6-30 MapReduce 编程过程

当程序员编写好上述各类后，在 main 方法里，按下述过程，依次指向各类即可，至于程序的输入输出，则在输入格式和输出格式里指定：

- Step1 实例化配置文件类
- Step2 实例化 Job 类
- Step3 指向输入格式类
- Step4 指向 Mapper 类
- Step5 指向 Partitioner 类
- Step6 指向 Reduce 类
- Step7 指向 OutputFormat 类
- Step8 提交任务

但是，很多程序员在自己编写 MR 程序或参阅别人编写的 MR 代码时，看到的代码好像并没有如此“复杂”，“好像”只需要编写 Map 类与 Reduce 类即可，这又是为何呢？

其实这主要是因为默认情况下，MapReduce 框架已经指定了各个类，表 6-2 为在用户不显示指定各类时，MapReduce 框架默认使用类。

在用户不做任何显示设置的情况下，MapReduce 框架就已经默认使用上述类了，故最简单情况下，用户只需要编写 Map 类和 Reduce 类，即可完成任务。

表 6-2 Yarn 框架处理 MR 程序时默认类

类 名	默认类
InputFormat	TextInputFormat
RecordReader	LineRecordReader
InputSplit	FileSplit
Map	IdentityMapper
Combine	不使用
Partitioner	HashPartitioner
GroupCompator	不使用
Reduce	IdentityReducer
OutputFormat	FileOutputFormat
RecordWriter	LineRecordWriter
OutputCommitter	FileOutputCommitter

6.6 实战 MapReduce 编程之 WordCount

单词计数是最简单也是最能体现 MapReduce 思想的程序之一，可以称为 MapReduce 版“Hello World”^[3]，该程序的完整代码可以在 Hadoop 安装包的 `src/examples` 目录下找到。单词计数主要完成的功能是：统计一系列文本文件中每个单词出现的次数，如图 6-31 所示。本小节将通过分析源代码帮助读者摸清 MapReduce 程序的基本结构。

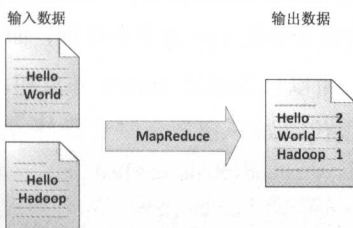


图 6-31 单词计数

6.6.1 WordCount 代码分析

MapReduce 框架自带的示例程序 WordCount 只包含 Mapper 类和 Reducer 类，其他全部使用默认类，下面为 WordCount 源码分析。

1. Mapper 类

Map 过程需要继承 `org.apache.hadoop.mapreduce` 包中 Mapper 类，并重写其 `map` 方法。通过在 `map` 方法中添加语句把 key 值和 value 值输出到控制台的代码，可以发现 `map` 方法中的 value 值存储的是文本文件中的一行（以回车符为行结束标记），而 key 值为该行的首字符相对于文本文件的首地址的偏移量。然后 `StringTokenizer` 类将每一行拆分成一个个的单词，并将 `<word, 1>` 作为 `map` 方法的结果输出，其余的工作都交由 MapReduce 框架处理。其中 `IntWritable` 和 `Text` 类是 Hadoop 对 `int` 和 `string` 类的封装，这些类能够被序列化，以方便在分布式环境中进行数据交换。`TokenizerMapper` 的实现代码如下：

```
public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    //map 方法，划分一行文本，读一单词写出一个<单词,1>
    public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
```

```

System.out.println("key = " + key.toString());    //添加查看 key 值
System.out.println("value = " + value.toString()); //添加查看 value 值
StringTokenizer itr = new StringTokenizer(value.toString());
while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one); //写出<单词,1>
}
}
}

```

2. Reducer 类

Reduce 过程需要继承 org.apache.hadoop.mapreduce 包中 Reduce 类，并重写其 reduce 方法。实际上，reduce 方法的输入参数 key 为单个单词，而 values 是由各 Mapper 上对应单词的计数值所组成的列表，所以只要遍历 values 并求和，即可得到某个单词出现的总次数。IntSumReduce 类的实现代码如下：

```

public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get(); //相当于<Hello,1><Hello,1>，将两个 1 相加
        }
        result.set(sum);
        context.write(key, result); //写出这个单词，和这个单词出现次数<单词，单词出现次数>
    }
}

```

3. 主函数

在 MapReduce 中，由 Job 对象负责管理和运行一个计算任务，并通过 Job 的一些方法对任务的参数进行相关的设置。此处设置了使用 TokenizerMapper 完成 Map 过程和使用 IntSumReduce 完成 Combine 和 Reduce 过程。还设置了 Map 过程和 Reduce 过程的输出类型：key 的类型为 Text，value 的类型为 IntWritable。任务的输入和输出路径则由命令行参数指定，并由 FileInputFormat 和 FileOutputFormat 分别设定。完成相应任务的参数设定后，即可调用 job.waitForCompletion() 方法执行任务。主函数实现代码如下：

```

public static void main(String[] args) throws Exception { //主方法，函数入口
    Configuration conf = new Configuration(); //实例化配置文件类
    Job job = new Job(conf, "WordCount"); //实例化 Job 类
    job.setInputFormatClass(TextInputFormat.class); //指定使用默认输入格式类
    TextInputFormat.setInputPaths(job, inputPaths); //设置待处理文件的位置
    job.setJarByClass(WordCount.class); //设置主类名
    job.setMapperClass(TokenizerMapper.class); //指定使用上述自定义 Map 类
}

```

```

job.setMapOutputKeyClass(Text.class);           //指定 Map 类输出的<K,V>, K 类型
job.setMapOutputValueClass(IntWritable.class);   //指定 Map 类输出的<K,V>, V 类型
job.setPartitionerClass(HashPartitioner.class);  //指定使用默认的 HashPartitioner 类
job.setReducerClass(IntSumReducer.class);        //指定使用上述自定义 Reduce 类
job.setNumReduceTasks(Integer.parseInt(numOfReducer)); //指定 Reduce 个数
job.setOutputKeyClass(Text.class);               //指定 Reduce 类输出的<K,V>,K 类型
job.setOutputValueClass(Text.class);             //指定 Reduce 类输出的<K,V>,V 类型
job.setOutputFormatClass(TextOutputFormat.class); //指定使用默认输出格式类
TextOutputFormat.setOutputPath(job, outputDir);  //设置输出结果文件位置
System.exit(job.waitForCompletion(true) ? 0 : 1); //提交任务并监控任务状态
}

```

4. 提交 WordCount

将上述代码片段合到一起，接着分别设置输入文件位置、输出文件位置、reduce 个数，可形成如下完整代码：

```

public class WordCount {
    //定义 map 类，一般继承自 Mapper 类，里面实现读取单词，写出<单词,1>
    public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        //map 方法，划分一行文本，读一个单词写出一个<单词,1>
        public void map(Object key, Text value, Context context) throws IOException,
        InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one); //写出<单词,1>
            }
        }
        //定义 reduce 类，对相同的单词，把它们<K,VList>中的 VList 值全部相加
        public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
            private IntWritable result = new IntWritable();
            public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
                int sum = 0;
                for (IntWritable val : values) {
                    sum += val.get(); //相当于<Hello,1><Hello,1>, 将两个 1 相加
                }
                result.set(sum);
                context.write(key, result); //写出这个单词，和这个单词出现次数<单词，单词出现次数>
            }
        }
        public static void main(String[] args) throws Exception { //主方法，函数入口
            Configuration conf = new Configuration(); //实例化配置文件类
            Job job = new Job(conf, "WordCount"); //实例化 Job 类
        }
    }
}

```

```

job.setInputFormatClass(TextInputFormat.class);    //指定使用默认输入格式类
TextInputFormat.setInputPaths(job, args[0]);        //设置待处理文件的位置
job.setJarByClass(WordCount.class);                //设置主类名
job.setMapperClass(TokenizerMapper.class);          //指定使用上述自定义 Map 类
job.setCombinerClass(IntSumReducer.class);          //指定开启 Combiner 函数
job.setMapOutputKeyClass(Text.class);               //指定 Map 类输出的<K,V>, K 类型
job.setMapOutputValueClass(IntWritable.class);      //指定 Map 类输出的<K,V>, V 类型
job.setPartitionerClass(HashPartitioner.class);     //指定使用默认的 HashPartitioner 类
job.setReducerClass(IntSumReducer.class);           //指定使用上述自定义 Reduce 类
job.setNumReduceTasks(Integer.parseInt(args[2]));  //指定 Reduce 个数
job.setOutputKeyClass(Text.class);                  //指定 Reduce 类输出的<K,V>,K 类型
job.setOutputValueClass(Text.class);                //指定 Reduce 类输出的<K,V>,V 类型
job.setOutputFormatClass(TextOutputFormat.class);   //指定使用默认输出格式类
TextOutputFormat.setOutputPath(job, new Path(args[1])); //设置输出结果文件位置
System.exit(job.waitForCompletion(true) ? 0 : 1);    //提交任务并监控任务状态
}

```

使用 Eclipse 开发工具将该代码打包, 假定打包后的文件名为 `hdpAction.jar`, 主类 `WordCount` 位于包 `njupt` 下, 则可使用如下命令向 YARN 集群提交本应用。

```

[allen@iclient0 ~]$ yarn jar hdpAction.jar njupt.WordCount /user/allen/in/ihadppy.txt
out/wc-00 4

```

其中“yarn”为命令, “jar”为命令参数, 后面紧跟打包后的代码地址, “njupt”为包名, “WordCount”为主类名, “/user/allen/in/ihadppy.txt”为输入文件在 HDFS 中的位置, “out/wc-00”为输出文件在 HDFS 中的位置, “4”为 Reduce 个数, 不过本题中若要保持结果文件数和图 6-31 相同, 则 Reduce 个数必须设为“1”。

6.6.2 WordCount 处理过程

上一节已经给出了 `WordCount` 的设计思路及源码, 但很多细节都未被提及, 本节将根据图 6-31 给出的处理过程, 对 `WordCount` 进行更详细的讲解。详细的执行步骤如下。

①将文件拆分成 splits, 由于测试用的文件较小, 所以每个文件为一个 split, 并将文件按行分割形成<key, value>对, 如图 6-32 所示。这一步由 MapReduce 框架自动完成, 其中偏移量 (即 key 值) 包括了回车所占的字符数 (Windows 和 Linux 环境下会不同)。

②将分割好的<key, value>对交给用户定义的 `map` 方法进行处理, 生成新的<key, value>对, 如图 6-33 所示。

③得到 `map` 方法输出的<key, value>对后, Mapper 会将它们按照 key 值进行排序, 并执行 Combine 过程, 将 key 值相同的 value 值累加, 得到 Mapper 的最终输出结果, 如图 6-34 所示。



图 6-32 分割过程

图 6-33 执行 map 方法

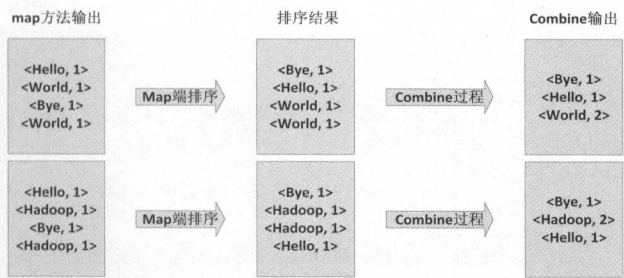


图 6-34 Map 端排序及 Combine 过程

④Reduce 先对从 Mapper 接收的数据进行排序，再交由用户自定义的 reduce 方法进行处理，得到新的<key, value>对，并作为 WordCount 的输出结果，如图 6-35 所示。

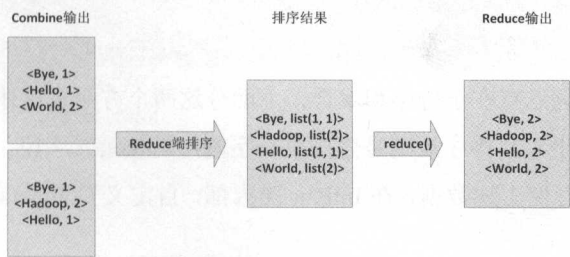


图 6-35 Reduce 端排序及输出结果

6.7 实战 MapReduce 编程之 SecondarySort

对如下输入数据集，假定一行为一个整体，比如“7 444”为一整体，“3 9999”也为一个整体。要求将第一字段相同的各行数据聚到一起，且对聚到一起后的数据，按第二字段排序。

7 444
3 9999

```

7 333
4 22
3 7777
7 555
3 6666
6 0
3 8888
4 11

```

比如对于上述数据, 第一字段只有“7”、“3”、“4”、“6”四种情况, 故其会聚成四类, 这四个类内部, 则按第二字段排序, 可得如下结果:

```

-----
3    6666
3    7777
3    8888
3    9999
-----
4    11
4    22
-----
6    0
-----
7    333
7    444
7    555

```

显然, 上述问题难点在于排序和聚合, 下面分这两个方面来讲述。

从结果集可看出, 其排序规则是先按第一字段, 后按第二字段。基于此, 此处自定义 `IntPair` 类, 用于存储上述数据, 在 `IntPair` 类内部, 自定义 `Comparator` 类以实现第一和第二字段一起比较。

显然, 按要求, 聚合的规则为第一字段, 为此自定义了 `FirstPartitioner` 类, 在 `FirstPartitioner` 内部, 指定聚合规则为第一字段。

此外, 为开启 MapReduce 框架自定义 `Partitioner` 功能, 须通过 `setPartitionerClass` 方法指定 `FirstPartitioner`。为开启 MapReduce 框架 `GroupingComparator` 功能, 须通过 `setGroupingComparatorClass` 方法指定 `FirstGroupingComparator`。项目完整代码如下:

```

package njupt;
public class SecondarySort {
    /**
     * Define a pair of integers that are writable.
     * They are serialized in a byte comparable format.
     */
    public static class IntPair implements WritableComparable<IntPair> {

```



```

private int first = 0;
private int second = 0;
public void set(int left, int right) {
    first = left;
    second = right;
}
public int getFirst() { return first; }
public int getSecond() { return second; }
/**
 * Read the two integers.
 * Encoded as: MIN_VALUE -> 0, 0 -> -MIN_VALUE, MAX_VALUE -> -1
 */
public void readFields(DataInput in) throws IOException {
    first = in.readInt() + Integer.MIN_VALUE;
    second = in.readInt() + Integer.MIN_VALUE;
}
public void write(DataOutput out) throws IOException {
    out.writeInt(first - Integer.MIN_VALUE);
    out.writeInt(second - Integer.MIN_VALUE);
}
public int hashCode() {
    return first * 157 + second;
}
public boolean equals(Object right) {
    if (right instanceof IntPair) {
        IntPair r = (IntPair) right;
        return r.first == first && r.second == second;
    } else { return false; }
}
/** A Comparator that compares serialized IntPair. */
public static class Comparator extends WritableComparator {
    public Comparator() { super(IntPair.class); }
    public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
        return compareBytes(b1, s1, l1, b2, s2, l2);
    }
}
// register this comparator
static { WritableComparator.define(IntPair.class, new Comparator()); }
public int compareTo(IntPair o) {
    if (first != o.first) {
        return first < o.first ? -1 : 1;
    } else if (second != o.second) {
        return second < o.second ? -1 : 1;
    } else { return 0; }
}

```

```

    }
}
/**
 * Partition based on the first part of the pair.
 */
public static class FirstPartitioner extends Partitioner<IntPair,IntWritable>{
    public int getPartition(IntPair key, IntWritable value, int numPartitions) {
        return Math.abs(key.getFirst() * 127) % numPartitions;
    }
}
/**
 * Compare only the first part of the pair, so that reduce is called once
 * for each value of the first part.
 */
public static class FirstGroupingComparator implements RawComparator<IntPair> {
    public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
        return WritableComparator.compareBytes(b1, s1, Integer.SIZE/8,
                                                b2, s2, Integer.SIZE/8);
    }

    public int compare(IntPair o1, IntPair o2) {
        int l = o1.getFirst();
        int r = o2.getFirst();
        return l == r ? 0 : (l < r ? -1 : 1);
    }
}
/**
 * Read two integers from each line and generate a key, value pair
 * as ((left, right), right).
 */
public static class MapClass extends Mapper<LongWritable, Text, IntPair, IntWritable> {
    private final IntPair key = new IntPair();
    private final IntWritable value = new IntWritable();
    public void map(LongWritable inKey, Text inValue,
                    Context context) throws IOException, InterruptedException {
        StringTokenizer itr = new StringTokenizer(inValue.toString());
        int left = 0;
        int right = 0;
        if (itr.hasMoreTokens()) {
            left = Integer.parseInt(itr.nextToken());
            if (itr.hasMoreTokens()) {
                right = Integer.parseInt(itr.nextToken());
            }
            key.set(left, right);
            value.set(right);
            context.write(key, value);
        }
    }
}

```

```

    }
}
}
/**
 * A reducer class that just emits the sum of the input values.
 */
public static class Reduce extends Reducer<IntPair, IntWritable, Text, IntWritable> {
    private static final Text SEPARATOR = new Text("-----");
    private final Text first = new Text();
    public void reduce(IntPair key, Iterable<IntWritable> values, Context context
        ) throws IOException, InterruptedException {
        context.write(SEPARATOR, null);
        first.set(Integer.toString(key.getFirst()));
        for(IntWritable value: values) {
            context.write(first, value);
        }
    }
}

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "secondary sort");
    job.setJarByClass(SecondarySort.class);
    job.setMapperClass(MapClass.class);
    job.setReducerClass(Reduce.class);
    job.setNumReduceTasks(Integer.parseInt(args[2]));
    // group and partition by the first int in the pair
    job.setPartitionerClass(FirstPartitioner.class);
    job.setGroupingComparatorClass(FirstGroupingComparator.class);
    // the map output is IntPair, IntWritable
    job.setMapOutputKeyClass(IntPair.class);
    job.setMapOutputValueClass(IntWritable.class);
    // the reduce output is Text, IntWritable
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}

```

使用 Eclipse 开发工具将该代码打包, 假定打包后的文件名为 `hdpAction.jar`, 主类 `SecondarySort` 位于包 `njupt` 下, 则可使用如下命令向 YARN 集群提交本应用。

```
[allen@iclient0 ~]$ yarn jar hdpAction.jar njupt.SecondarySort /user/allen/in/ss.txt out/ss-004
```

其中“yarn”为命令，“jar”为命令参数，后面紧跟打包后的代码地址，“njupt”为包名，“SecondarySort”为主类名，“/user/allen/in/ss.txt”为本节开头时给的输入文件在 HDFS 中的绝对路径，“out/wc-00”为输出文件在 HDFS 中的相对路径，“4”为 Reduce 个数。

6.8 实战 MapReduce 编程之倒排索引

倒排索引在搜索领域应用非常广，本节即讲述使用 MapReduce 来构建倒排索引。

6.8.1 简介

倒排索引^[4]是文档检索系统中最常用的数据结构，被广泛地应用于全文搜索引擎。它主要用来存储某个单词（或词组）在一个文档或一组文档中的存储位置的映射，即提供了一种根据内容来查找文档的方式。由于不是根据文档来确定文档所包含的内容，而是进行了相反的操作，因而称为倒排索引（Inverted Index）。通常情况下，倒排索引由一个单词（或词组）以及相关的文档列表组成，文档列表中的文档或者标识文档的 ID 号，或者指定文档所在位置的 URI，如图 6-36 所示。

从图 6-36 可以看出，单词 1 出现在{文档 1，文档 4，文档 13，……}中，单词 2 出现在{文档 3，文档 5，文档 15，……}中，而单词 3 出现在{文档 1，文档 8，文档 20，……}中。在实际应用中，还需要给每个文档添加一个权值，用来指出每个文档与搜索内容的相关度，如图 6-37 所示。

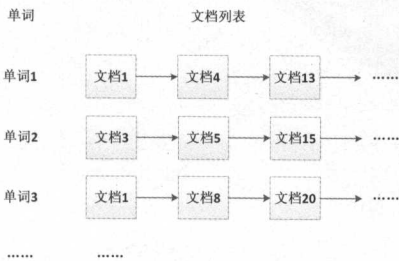


图 6-36 倒排索引结构

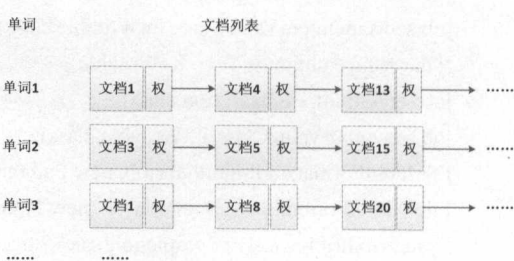


图 6-37 添加权重的倒排索引

最常用的是使用词频作为权重，即记录单词在文档中出现的次数。以英文为例，如图 6-38 所示，索引文件中的“MapReduce”一行表示：“MapReduce”这个单词在文本 T0 中出现过 1 次，T1 中出现过 1 次，T2 中出现过 2 次。当搜索条件为“MapReduce”、

“is”、“simple”时，对应的集合为： $\{T0, T1, T2\} \cap \{T0, T1\} \cap \{T0, T1\} = \{T0, T1\}$ ，即文本 T0 和 T1 包含了所要索引的单词，而且只有 T0 是连续的。

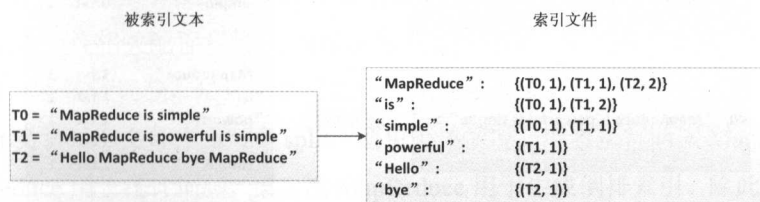


图 6-38 倒排索引示例

更复杂的权重还可能要记录单词在多少个文档中出现过，以实现 TF-IDF (Term Frequency-Inverse Document Frequency) 算法，或者考虑单词在文档中的位置信息（单词是否出现在标题中，反映了单词在文档中的重要性）等。本节将针对英文文本，使用词频作为权重，讲解如何使用 MapReduce 分析、设计和实现倒排索引。

6.8.2 分析与设计

本节实现的倒排索引主要关注的信息为：单词、文档 URI 及词频，如图 6-38 所示。但是在实现过程中，索引文件的格式与图 6-38 会略显不同，以避免重写 OutputFormat 类。下面根据 MapReduce 的处理过程给出倒排索引的设计思路。

1. Map 过程

首先使用默认的 TextInputFormat 类对输入文件进行处理，得到文本中每行的偏移量及其内容。显然，Map 过程首先必须分析输入的 <key, value> 对，得到倒排索引中需要的三个信息：单词、文档 URI 和词频，如图 6-39 所示。这里存在两个问题：第一，<key, value> 对只能有两个值，在不使用 Hadoop 自定义数据类型的情况下，需要根据情况将其中两个值合并成一个值，作为 key 或 value 值；第二，通过一个 Reduce 过程无法同时完成词频统计和生成文档列表，所以必须增加一个 Combine 过程完成词频统计。

这里将单词和 URI 组成 key 值（如 “MapReduce:1.txt”），将词频作为 value，这样做的好处是可以利用 MapReduce 框架自带的 Map 端排序，将同一文档的相同单词的词频组成列表，传递给 Combine 过程，实现类似于 WordCount 的功能。

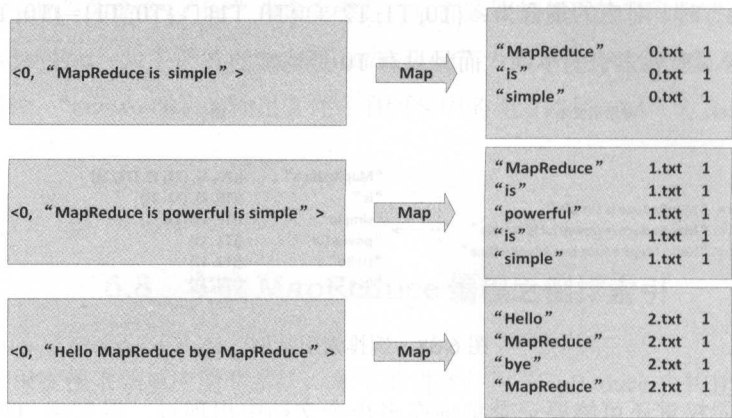


图 6-39 Map 过程输入/输出

2. Combine 过程

经过 map 方法处理后，Combine 过程将 key 值相同的 value 值累加，得到一个单词在文档中的词频，如图 6-40 所示。如果直接将图 6-40 所示的输出作为 Reduce 过程的输入，在 Shuffle 过程时将面临一个问题：所有具有相同单词的记录（由单词、URI 和词频组成）应该交由同一个 Reduce 处理，但当前的 key 值无法保证这一点，所以必须修改 key 值和 value 值。这次将单词作为 key 值，URI 和词频组成 value 值（如“1.txt:1”）。这样做的好处是可以利用 MapReduce 框架默认的 HashPartitioner 类完成 Shuffle 过程，将相同单词的所有记录发送给同一个 Reduce 进行处理。

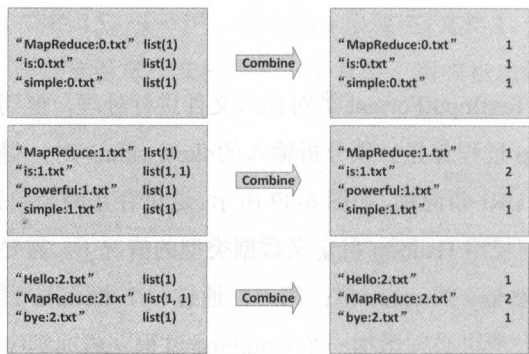


图 6-40 Combine 过程输入/输出

3. Reduce 过程

经过上述两个过程后，Reduce 过程只需要将相同 key 值的 value 值组合成倒排索引文件所需的格式即可，剩下的事情就可以直接交给 MapReduce 框架进行了，如图 6-41 所示。索引文件的内容除分隔符外与图 6-38 解释相同。

4. 需要解决的问题

本节设计的倒排索引在文件数目上没有限制，但是单个文件不宜过大（具体值与默认 HDFS 块大小及相关配置有关），要保证每个文件对应一个 split。否则，由于 Reduce 过程没有进一步统计词频，最终结果可能会出现词频未统计完全的单词。可以通过重写 InputFormat 类将每个文件作为一个 split，避免上述情况。或者执行两次 MapReduce，第一次 MapReduce 用于统计词频，第二次 MapReduce 用于生成倒排索引。除此之外，还可以利用复合键值对等实现包含更多信息的倒排索引。读者可以尝试修改或优化本节给出的代码。

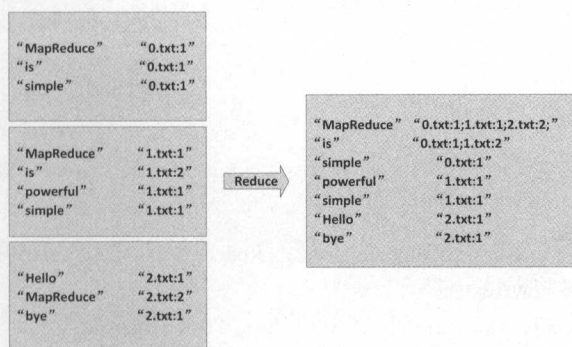


图 6-41 Reduce 过程输入/输出

6.8.3 倒排索引完整源码

根据 6.8.2 节的分析而编写的倒排索引完整源代码如下所示，对代码的详细分析以注释的形式给出。

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reduce;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.FileSplit;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;
public class InvertedIndex {
    public static class InvertedIndexMapper extends
```



```

Mapper<Object, Text, Text, Text>{
    private Text keyInfo = new Text();//存储单词和 URI 的组合
    private Text valueInfo = new Text();//存储词频
    private FileSplit split; //存储 Split 对象
    public void map(Object key, Text value, Context context) throws IOException,
InterruptedException{
        //获得<key, value>对所属的 FileSplit 对象
        split = (FileSplit)context.getInputSplit();
        StringTokenizer itr = new StringTokenizer(value.toString());
        while(itr.hasMoreTokens()){
            //key 值由单词和 URI 组成, 如 "MapReduce:1.txt"
            keyInfo.set(itr.nextToken() + ":" + split.getPath().toString());
            //词频初始为 1
            valueInfo.set("1");
            context.write(keyInfo, valueInfo);
        }
    }
}

public static class InvertedIndexCombiner extends Reduce<Text, Text, Text, Text>{
    private Text info= new Text();
    public void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException{
        //统计词频
        int sum = 0;
        for(Text value : values){
            sum += Integer.parseInt(value.toString());
        }
        int splitIndex = key.toString().indexOf(":");
        //重新设置 value 值由 URI 和词频组成
        info.set(key.toString().substring(splitIndex + 1) + ":" + sum);
        //重新设置 key 值为单词
        key.set(key.toString().substring(0, splitIndex));
        context.write(key, info);
    }
}

public static class InvertedIndexReduce extends Reduce<Text, Text, Text, Text>{
    private Text result = new Text();
    public void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException{
        //生成文档列表
        String fileList = new String();
        for(Text value : values){
            fileList += value.toString() + ",";
        }
    }
}

```

```

        result.set(fileList);
        context.write(key, result);
    }
}

public static void main(String[] args) throws Exception{
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if(otherArgs.length != 2){
        System.err.println("Usage: invertedindex <in> <out>");
        System.exit(2);
    }
    Job job = new Job(conf, "InvertedIndex");
    job.setJarByClass(InvertedIndex.class);
    job.setMapperClass(InvertedIndexMapper.class);
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(Text.class);
    job.setCombinerClass(InvertedIndexCombiner.class);
    job.setReduceClass(InvertedIndexReduce.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(Text.class);
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}

```

MapReduce 并行计算框架为程序员屏蔽了很多底层的处理细节，简化了编程过程，减轻了程序员的负担，提高了编程效率，对于很多计算问题，程序员通常只要使用默认设置，仅仅设计实现 Map 和 Reduce 这两个函数即可。为了增强 MapReduce 的适应性，Hadoop MapReduce 并行编程框架还提供了很多丰富而灵活的处理机制和高级编程技术，程序员可以使用这些高级编程技术和方法，设计实现各种复杂的计算问题。本章将介绍一系列高级编程技巧和方法。

6.9 实战 MapReduce 之性能优化

由于已经有了 MRAppMaster 和 MRClient，当程序员试图编写 MR-App 时只需要编写 MRAppBusinessLogic。从 MRAppBusinessLogic 编写步骤看出，其编写过程非常简单，甚至多数情况下只需要编写 Mapper 类和 Reduce 类，这么简单的编程模型，何来优化

可言。

实际上，这是由于 MapReduce 框架默默地为 MRAppBusinessLogic 做了太多的事情，才让我们误以为其简单。试想一下，一个能够将普通程序拓展到上万台机器上同时执行的应用框架，怎么可能会简单。所谓的 MR-App 调优，就是指根据硬件环境、OS 环境、业务逻辑适当的调整 MapReduce 的默认设置，有时也会在 MRAppBusinessLogic 代码中直接读取第三方系统，以加快存取速度。下面编者先汇总常见的 MapReduce 调优，接着在每一小节具体讲述一种调优方法（表 6-3）。

表 6-3 MR-App 调优方法汇总

默认情况	调优方法	注意点
默认块太小	调整块大小	受限于硬件
默认不开启二进制	开启二进制输入输出	完美
默认只用一个 Reduce	设置最佳 Reduce 数量	受限于集群大小
默认不开启多线程 Map	启用多线程 Map	受限于业务类型
默认不开启分布式缓存	开启分布式缓存	受限于业务类型
默认值较优	增大排序时内存空间	不常用
默认不开启 Combine	开启 Combine	受限于业务类型
默认不开启第三方存取	读取第三方存取	受限于业务类型
默认不够用	自定义数据类型	完美

下面对表 6-3 中给出的优化方法，逐一讲述。

1. 增加块大小

默认分片大小就是块大小，在 Hadoop1.0 中，块大小默认为 64M，Hadoop2.0 中则增大到 128M。如果服务器性能较优，建议将块大小再次上调到 256M。比如对于内存 32G，CPU16 核的单机来说，处理 128M 需要开启一个进程，处理 256M 也要开启一个进程，而进程的启动、切换和销毁开销很大，非常不值。

以 WordCount 为例，此设置适用的前提是 WordCount 处理的数据量非常大，一般都在 1T 以上；单机硬件性能较优，比如单机内存 32G，CPU16 核；集群规模中等，常见的节点在一百到一千之间。

比如对于单机标配“32G、16 核”的 200 节点 Hadoop 集群，当其要处理 1T 数据时，可采用下述方法设置其块大小。

由于单机 32G、16 核，最好在该机上启动 8 个单进程，每个进程持有一个核、4G 内存。又由于 200 个节点， $200 \times 8 = 1600$ 。故整个集群中启动 1600 个进程时，对硬件来说性能较优， $1T / 1600 = 1024 \times 1024M / 1600 = 655.36$ 。故“块”大小设置为 512M 时，性能较优。

设置集群“块”非常简单，只需要将下述内容添加到集群中所有机器的 hdfs-site.xml

文件里。在 littleCstor 中，可在 Ambari 主页面上 HDFS 里设置该参数。

```
<property>
  <name>dfs.blocksize</name>
  <value>512m</value>
</property>
```

该设置的缺点是，集群重启才能生效，对于已经上传的数据，块大小依旧是上传之前的块大小，其只对重启后上传的数据生效。

2. 开启二进制输入输出

计算机中，一切的计算和存取最终都要转换成二进制，MapReduce 框架默认的 Text 格式只是给程序员“眼睛”看的，实际上 Text 格式严重降低了单机性能。除了调试，二进制输入输出在任何情况下都非常适用，其设置方式也非常简单，只在 main 方法里指定即可，下面以 WordCount 为例，讲述其开启二进制输入输出方式：

```
public class WordCount {
    public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        public void map(Object key, Text value, Context context) throws IOException,
        InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
        public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
            private IntWritable result = new IntWritable();
            public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
                int sum = 0;
                for (IntWritable val : values) {sum += val.get();}
                result.set(sum);
                context.write(key, result);
            }
        }
        public static void main(String[] args) throws Exception {
            Configuration conf = new Configuration();
            Job job = new Job(conf, "word count");
            job.setJarByClass(WordCount.class);
            job.setMapperClass(TokenizerMapper.class);
            job.setCombinerClass(IntSumReducer.class);
            job.setReducerClass(IntSumReducer.class);
            job.setOutputKeyClass(Text.class);
```



```

        job.setOutputValueClass(IntWritable.class);
job.setInputFormatClass(TextInputFormat.class); //显然没有 sequence 格式输入文件故此处只能 text
        TextInputFormat.addInputPaths(job, "in"); //输入文件为 text 格式
job.setOutputFormatClass(SequenceFileOutputFormat.class); //SequenceFileOutputFormat 输出
SequenceFileOutputFormat.setOutputPath(job, new Path(args[1])); //输出文件为 Sequence 格式
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}

```

显然，之前没有二进制文件，故无二进制文件可读取，只能设置二进制输出；上述程序执行后，就有了一个二进制文件，此时只需要将 main 方法调整如下，即可读取二进制输入：

```

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    job.setInputFormatClass(SequenceFileInputFormat.class); //SequenceFileInputFormat 输入
    SequenceFileInputFormat.addInputPaths(job, "in"); //输入文件须为 Sequence 格式
    job.setOutputFormatClass(SequenceFileOutputFormat.class); //SequenceFileOutputFormat 输出
    SequenceFileOutputFormat.setOutputPath(job, new Path("out")); //输出文件为 Sequence 格式
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}

```

依编者经验，开启 SequenceFileInputFormat 和 SequenceFileOutputFormat 后，对于单机标配“32G、16 核”的 200 节点 Hadoop 集群，扫描 1T 数据时，性能可以提升一倍。

进一步，程序员可开启存取和处理效率更高的 SequenceFileAsBinaryInputFormat 和 SequenceFileAsBinaryOutputFormat，请读者自行完成。

3. 开启最佳 Reduce 数量

在 MapReduce 框架中，Reduce 个数是由业务逻辑（程序员）决定的，那么，当业务逻辑允许使用多个 Reduce 时，到底使用多少个 Reduce 合适呢？答案是，由集群节点数决定，比如对于单机标配“32G、16 核”的 200 节点 Hadoop 集群，最佳的 Reduce 个数是 200 个。

这是因为当每个节点上都开启一个 Reduce 时，只要充分设置该 Reduce 核数、内存量。可实现网络阶段的 Shuffle 充分并行，200 个 Reduce 并行，机器性能方面，单机上的 Reduce 都能达到最佳。Reduce 个数设置起来非常简单，在 main 方法，使用如下一行代码即可，下面以 WordCount 为例，设置 WordCount 启用多个 Reduce：

```

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setNumReduceTasks(200); // 设置 Reduce 个数为 200
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path("in")); // 设置输入路径为相对路径 in
    FileOutputFormat.setOutputPath(job, new Path("out")); // 设置输出路径为相对路径 out
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}

```

特别的，对于存在“数据倾斜”型的 MR-App，显然无论设置多少 Reduce 都是没用的，此时建议采用两步走策略，首先，将业务逻辑写成多个 MapReduce 串，比如某任务为 MR-App1→MR-App2→MR-App3；然后对于 MR-App1 和 MR-App2，重写其 HashPartitioner，让前两个 MR-App 数据尽量均散开，实现大数据不停规约成小数据，这样对于第三个 MR-App3，即使还是发生倾斜，其数据量已经小了很多。

4. 启用多线程 Mapper

当程序员要编写的 MR-App 业务逻辑上属于 CPU 密集型（如机器学习）任务时，建议将普通的单线程 Map 换成 MultithreadedMapper 或 MultithreadedMapRunner，这两个都是 Map 类的多线程实现。

需要指出的是，当开启 MultithreadedMapper 时，应根据集群规模，严格限制 Mapper 个数，比如对于单机标配“32G、16 核”的 200 节点 Hadoop 集群，最佳方案是，同一时刻，集群中只有 200 个 Mapper（在同时运行）。这里说的不是该 MR-App 有多少 Mapper，而是集群中同时执行的 Mapper 数，该 MR-App 大可持有 1000 个 Mapper，但集群同时运行的最好是 200 个。

这是因为对于“32G、16 核”的单机来说，若其上只运行一个 MultithreadedMapper，该进程会充分利用 CPU 和内存资源，若该单机上同时运行多个 MultithreadedMapper，极有可能会出现内存不足，CPU 不够等情况，从而造成大量内存换进换出，甚至是抖动。要知道，MultithreadedMapper 很耗内存和 CPU。

当集群中同时运行 200 个 Mapper 时，默认每个节点上只会启动一个 MultithreadedMapper，此时性能最佳。

5. 开启分布式缓存

假定有一个大小为 1T 的大数据集 datasetB1 和一个 100M 的小数据集 datasetZ1，判断 datasetZ1 中的数据是否在 datasetB1 中，借助分布式缓存，可实现并行处理，步骤如下：

Step1 主程序告知各 Mapper，datasetZ1 位置。

Step2 各 Mapper 加载 datasetZ1 至内存中的 HashSet 内。

Step3 各 Mapper 顺序扫描 datasetB1 每行数据并判断该行是否存在 HashSet 中，存在则输出，不存在，则不输出。

显然上述过程并不需要 Reduce，在 Mapper 中更为合适，下面为上述步骤对应的代码清单：

```
package njupt;
import java.io.IOException;
import java.util.HashSet;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
import org.apache.hadoop.mapreduce.lib.partition.HashPartitioner;
import org.apache.hadoop.util.LineReader;
public class DistributedCache {
    public static class TokenizerMapper extends Mapper<Object, Text, Text, Text> {
        HashSet<String> littleSet;
        public void setup(Mapper<Object, Text, Text, Text>.Context context)
            throws IOException, InterruptedException {
            super.setup(context);
            littleSet=new HashSet<String>();//用来存储小数据集
            //获取 main 传来的小表位置，注意该文件系统为 HDFS
            Path datasetZ1=new Path(context.getConfiguration().get("datasetZ1Location"));
            FileSystem hdfs=datasetZ1.getFileSystem(context.getConfiguration());//获取 HDFS 句柄
            FSDataInputStream fsInputStream=hdfs.open(datasetZ1);//创建输入流
            LineReader lReader=new LineReader(fsInputStream);//顺序读取行
            Text line = null;
            while (lReader.readLine(line)!=0) { //将小表中的各行加入 HashSet
                littleSet.add(line.toString());
            }
        }
    }
}
```



```

public void map(Object key, Text value, Context context) throws IOException,
InterruptedException {
    String bigDataSetLine=value.toString();//顺序扫描大表, bigDataSetLine 为大表中一行
    if (littleSet.contains(bigDataSetLine)) { //判断该行是否在 HashSet 中
context.write(new Text(value.toString()), new Text("exist"));//bigDataSetLine 在小表中, 输出
    }
}
}

public static void main(String[] args) throws Exception { //主方法, 函数入口
    Configuration conf = new Configuration(); //实例化配置文件类
    Job job = new Job(conf, "DistributedCache"); //实例化 Job 类
    job.setInputFormatClass(TextInputFormat.class); //指定使用默认输入格式类
    TextInputFormat.setInputPaths(job, args[0]); //设置大数据集 datasetB1 在 HDFS 中的位置
    job.setJarByClass(DistributedCache.class); //设置主类名
    job.setMapperClass(TokenizerMapper.class); //指定使用上述自定义 Map 类
    job.setMapOutputKeyClass(Text.class); //指定 Map 类输出的<K,V>, K 类型
    job.setMapOutputValueClass(Text.class); //指定 Map 类输出的<K,V>, V 类型
    //job.setPartitionerClass(HashPartitioner.class); //不开启 Partitioner
conf.set("datasetB1Location", "/user/allen/datasetB1");//main 方法向 Mapper 传递 datasetZ1 位置
    //job.setReducerClass(IntSumReducer.class); //不开启 Reduce
    job.setNumReduceTasks(0); //不开启 Reduce 时, 必须指定 Reduce 个数为 0
    job.setOutputFormatClass(TextOutputFormat.class); //指定使用默认输出格式类
    TextOutputFormat.setOutputPath(job, new Path(args[1])); //设置输出结果文件位置
    System.exit(job.waitForCompletion(true) ? 0 : 1); //提交任务并监控任务状态
}
}

```

由于执行 main 方法的 MRAppMaster 和执行 Mapper 类的 YarnChild 都为独立进程, 其不能通过共享变量方式传递共享变量, 只能通过 main 方法里设置, Mapper 或 Reduce 里获取, 这种方式:

main 方法里设置:

```
conf.set("datasetB1Location", "/user/allen/datasetB1");
```

Mapper 或 Reducer 里获取

```
String datasetB1=context.getConfiguration().get("datasetB1Location");
```

此外, 从上述程序中还可看出, MR-App 完全可以只有 Map 而无 Reduce。

6. 增大排序时内存空间

在 Mapper 使用 map()方法将数据处理结束后, MapReduce 框架会对这些处理后的数据按 Key 排序。排序过程中, 若数据量正常, 框架会使用内排序, 若数据量太大, 框架会使用外排序。

在 Reducer 执行 reduce()方法之前, 框架会将汇总至本 Reduce 的数据进行排序, 排序算法和上述过程类似。

显然, 当你的业务中发生数据倾斜时, 局部排序完全变成了全部排序, 此时即可增大排序内存。

7. 开启 Combine

实际上, Combine 函数就是 Reduce 函数, 故开启 Combine 时能够大大规约数据集, 减轻 Shuffle 阶段带宽压力。Combine 的开启方式非常简单, 只需调用 `setCombinerClass` 即可:

```
job.setMapperClass(TokenizerMapper.class);
job.setCombinerClass(IntSumReducer.class);
job.setReducerClass(IntSumReducer.class);
```

需要提出的是, 由于 Combine 函数就是 Reduce 函数, 故其对业务逻辑要求很高, 当开启 combine 时, Mapper 的输出必须可规约, 否则 MR-App 的结果是错误的, 在编者编写 MR-App 过程中, 见过许多 MR-App 出错的原因都是设置了 Combine, 请慎用。

8. 使用 Secondarysort

6.7 节已经给出了 SecondarySort 示例代码, 充分利用 SecondarySort 的编程机制, 能够极大提高 MapReduce 编程灵活性。请读者结合 6.7 节代码, 深入分析 `IntPair` 定义和作用、`FirstPartitioner` 定义和作用、`FirstGroupingComparator` 定义和作用。

9. 自定义数据类型

对于 MapReduce 处理的 `<key,value>` 对, 有时并不能满足业务需求, 此时就需要用户自定义 key, value 类型, 实际上, 前面已经讲述了自定义 keyvalue 类型。显然 SecondarySort 里所述的 `IntPair` 就是这样的—个数据结构, 由于自定义数据类型非常重要, 下面编者再次给出 `IntPair` 源码。

1) 自定义 Key

key 必须实现 `WritableComparable` 接口, 对于此接口的自定义数据类型, 必须重写其 “`readFields`”、“`write`” 和 “`compareTo`” 方法。

```
package njupt;
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;
public class IntPair implements WritableComparable<IntPair> {
    private int first = 0;
    private int second = 0;
    /** Set the left and right values.*/
    public void set(int left, int right) {
        first = left;
```

```

        second = right;
    }
    public int getFirst() {return first;}
    public int getSecond() {return second;}
    //Read the two integers. Encoded as: MIN_VALUE -> 0, 0 -> -MIN_VALUE,
    //MAX_VALUE-> -1
    public void readFields(DataInput in) throws IOException {
        first = in.readInt() + Integer.MIN_VALUE;
        second = in.readInt() + Integer.MIN_VALUE;
    }
    public void write(DataOutput out) throws IOException {
        out.writeInt(first - Integer.MIN_VALUE);
        out.writeInt(second - Integer.MIN_VALUE);
    }
    public int hashCode() {return first * 157 + second;}
    public boolean equals(Object right) {
        if (right instanceof IntPair) {
            IntPair r = (IntPair) right;
            return r.first == first && r.second == second;
        } else {return false;}
    }
    /** A Comparator that compares serialized IntPair. */
    public static class Comparator extends WritableComparator {
        public Comparator() {super(IntPair.class);}
        public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
            return compareBytes(b1, s1, l1, b2, s2, l2);
        }
    }
    static { // register this comparator
        WritableComparator.define(IntPair.class, new Comparator());}
    public int compareTo(IntPair o) {
        if (first != o.first) {
            return first < o.first ? -1 : 1;
        } else if (second != o.second) {
            return second < o.second ? -1 : 1;
        } else {return 0;}
    }

```

从 IntPair 源码中可以看出，key 必须实现 WritableComparable 接口的“readFields”、“write”和“compareTo”方法，MapReduce 框架在网络传输过程中，须使用此类来序列化数据。

2) 自定义 Value

Value 只需要实现 Writable 接口，对于实现此接口的自定义数据类型，必须重写其“readFields”和“write”方法。

```

package njupt;
import java.io.DataInput;

```

```

import java.io.DataOutput;
import java.io.IOException;
import org.apache.hadoop.io.Writable;
public class StringInt implements Writable {
    private String first = "";
    private int second = 0;
    public void set(String left, int right) {
        first = left;
        second = right;
    }
    public String getFirst() {return first;}
    public int getSecond() {return second;}
    public void readFields(DataInput in) throws IOException {
        first = in.readUTF();
        second = in.readInt();
    }
    public void write(DataOutput out) throws IOException {
        out.writeUTF(first);
        out.writeInt(second);
    }
    public int hashCode() {return first.hashCode()+ second;}
    public boolean equals(Object right) {
        if (right instanceof StringInt) {
            StringInt r = (StringInt) right;
            return r.first.equals(first) && r.second == second;
        } else {return false;}}
}

```

从 StringInt 代码中可以看出, write()和 readFields()方法的写入和读取是对称的, 这点请程序员务必注意。

总之 MapReduce 调优是一个细致的工作, 其涉及 OS、硬件、集群规模、数据结构、业务逻辑等方面, 须深入分析才可发生质变。

习 题

1. “大数据”给计算带来了哪些挑战?
2. 简述 M-S-R 范式并行化步骤, 试问步骤之间和步骤内部是否也是并行的?
3. 简述 MapReduce 框架功能作用及其体系架构。
4. 简述手工部署 MapReduce、使用 Ambari 部署 MapReduce 的部署步骤。
5. 简述 MapReduce 访问接口。
6. 简述使用 Maven 时, MapReduce 开发环境搭建步骤, 不使用 Maven 时又如何?

7. 简述 MapReduce 编程步骤。
8. 简述在 YARN 上, MapReduce-App 执行步骤。
9. 在大型系统中, 如何使用 MapReduce 来提供在线和离线服务。
10. 在大型系统中, 如何使用 MapReduce 来处理增量数据?
11. 简述常见的 MapReduce-App 调优技术。
12. 简述 YARN-App 执行步骤。

参考文献

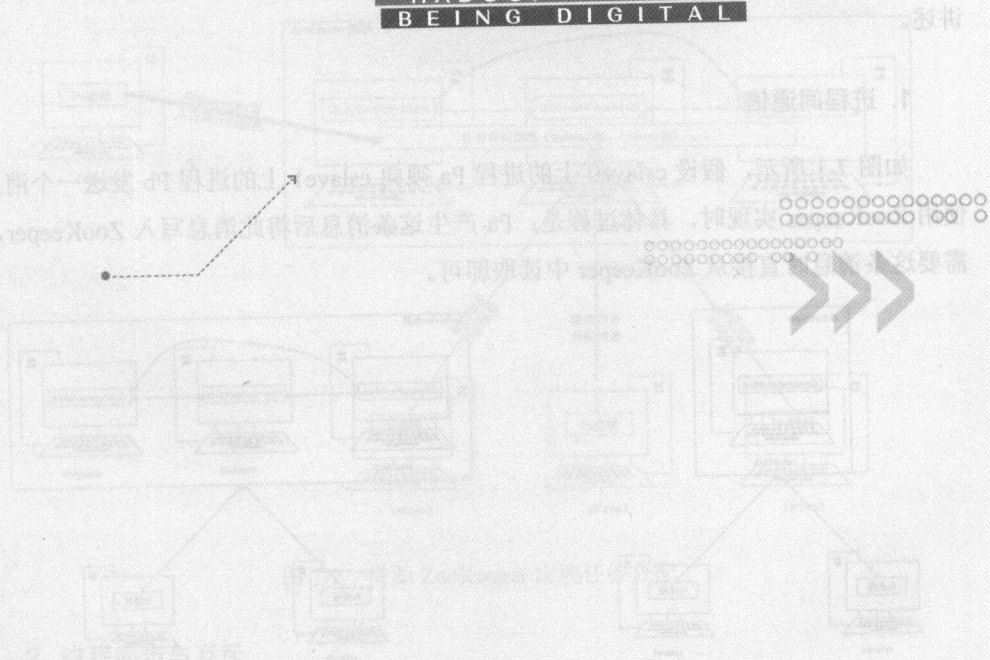
- [1] <http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html>
- [2] 刘鹏. 云计算[M]. 3 版. 北京: 电子工业出版社, 2015.
- [3] <http://wiki.apache.org/hadoop/WordCount>
- [4] https://en.wikipedia.org/wiki/Reverse_index

第7章

分布式锁服务 ZooKeeper

HADOOP

BEING DIGITAL



当单机上两个进程需要使用同一个资源时，即是所谓的“互斥”。在分布式环境下，不同机器上的多个进程间也存在着大量“同步”与“互斥”^[1]操作，ZooKeeper^[2]即是这样一个用来协调分布式环境下不同进程间“同步”与“互斥”操作的分布式锁服务。本章在讲述 ZooKeeper 基本知识后，将重点讲述借助 ZooKeeper 实现分布式环境下进程间通信。

7.1 ZooKeeper 简介

ZooKeeper（又称分布式锁）是由开源组织 Apache 开发的一个高效、可靠的分布式协调服务。从功能上看它是谷歌 Chubby 的开源实现（实现算法上存在细微区别），起初由雅虎开发，后来于 2008 年捐赠给 Apache，当前稳定版为 3.4.7。

7.1.1 ZooKeeper 应用场景

ZooKeeper 的经典应用场景是实现进程间通信（少量数据）和进程同步，下面分别讲述。

1. 进程间通信

如图 7-1 所示，假设 cslave0 上的进程 Pa 须向 cslave1 上的进程 Pb 发送一个消息，使用 ZooKeeper 实现时，具体过程是：Pa 产生这条消息后将此消息写入 ZooKeeper，Pb 需要这条消息时直接从 ZooKeeper 中读取即可。

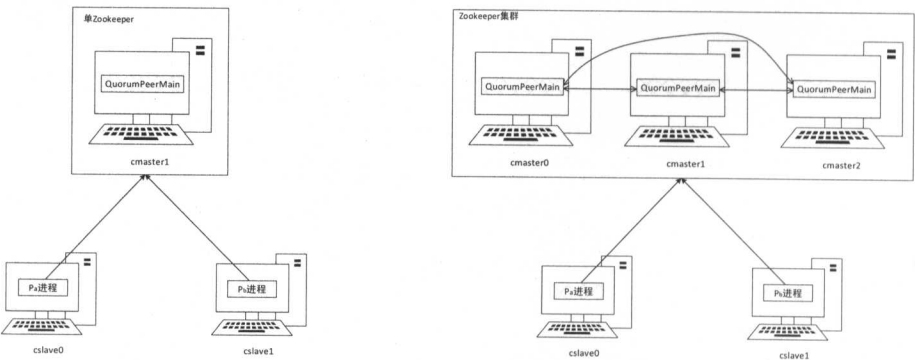


图 7-1 借助 ZooKeeper 实现进程通信

从此工作过程可以看出 ZooKeeper 提供了松耦合交互方式，即交互双方不必同时存在，也不用彼此了解。比如 Pa 在 ZooKeeper 中留下一条消息后，进程 Pa 结束，此后进程 Pb 才刚开始启动。

值得注意的是 ZooKeeper 服务本身也是不可靠的，比如运行 ZooKeeper 服务的机器宕机，则此服务失效，为提高 ZooKeeper 可靠性，在使用时 ZooKeeper 本身一般都以集群方式存在（图 7-1 右图），其内部实现将在体系架构一节讲述。

应当指出，这种数据交换方式适用于交换少量“核心数据”。若（运行于两台不同机器的）两个进程需要交换的信息量巨大，则一般使用 RPC 或消息传递机制，让这两个进程直接交换大量数据。比如“Pa 向 ZooKeeper 写入本进程结束，Pb 可以开始这一场景时”，就可以借助 ZooKeeper 来协调 Pa、Pb 执行次序。倘若 Pa 需要向 Pb 发送大量数据时，则最好是封装 Netty，让它们之间直接交换数据。

借助 ZooKeeper 实现进程通信的最经典应用场景是任务分配，以图 7-2 为例，此时 Pz 将任务分配信息写入 ZooKeeper 存储空间（至于到底存在哪个机器上，ZooKeeper 自己解决），Pa、Pb、Pc 读取 ZooKeeper 后即可知道本进程应当执行哪些任务。实际上，Storm 中，Nimbus 对任务的分配信息就是写入 ZooKeeper，各 Worker 查看该 ZooKeeper 即知本 Worker 需要执行哪些任务；而 Executor 之间交换大量数据时，则是借助 Netty 或 ZMQ 直接交换。

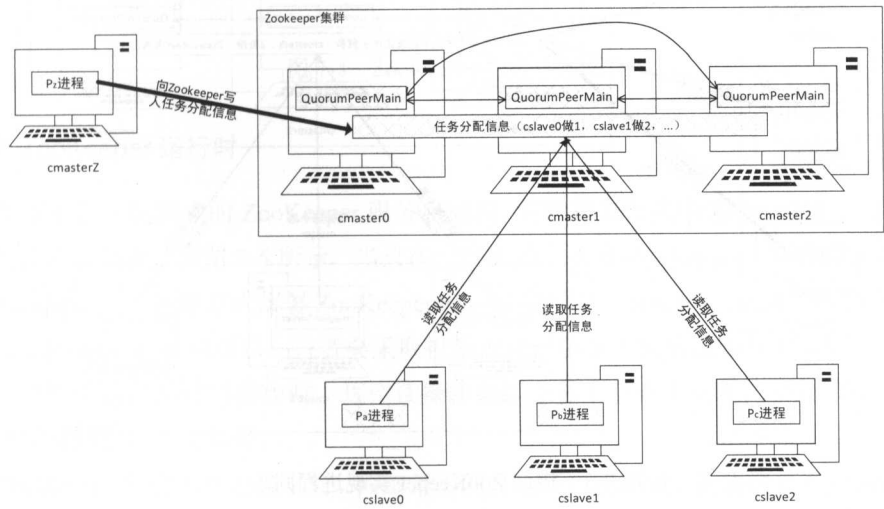


图 7-2 借助 ZooKeeper 实现任务分配

2. 进程同步与互斥

当多个进程间发生“同步”与“互斥”操作时，显然需要第三方来协调才能保证其

正确执行。此时,正在占用资源的进程可向 ZooKeeper 中写入互斥锁,待其使用结束后,再解除该互斥锁。当其他进程需要使用该资源时,其也向 ZooKeeper 写入互斥锁,倘若互斥锁已被锁定,其只能等待,待其解除该进程再占用该互斥锁。

比如打印机向 ZooKeeper 注册了打印机变量 iPrint,进程 Pa、Pb 同时要求使用该打印机,此时可以让 Pb 先在 iPrint 上设置互斥锁,待其打印结束后,Pb 再占用该锁。

借助 ZooKeeper 实现进程“同步”与“互斥”的经典场景是“BSP 范式下的进程同步”和“解决分布式环境下的单点故障”,下面以“单点故障为例”,讲述 ZooKeeper 的应用。

由于 HBase 采用 master/slave 架构(HMaster 为主服务,HRegionServer 为从服务),显然,当 master 发生故障时,HBase 服务将受到限制(此时的 HBase 只支持读不支持写),即 HBase 存在单点故障。此时,可以在集群中部署多个 HMaster 来解决单点故障问题,如图 7-3 所示,该 HBase 集群中有三个 HMaster,不过 ZooKeeper 集群中只有一把 HBase 互斥锁,集群启动时,这三个 HMaster 都会向 ZooKeeper 集群注册自己,显然只有一个 HMaster 能够抢占到该互斥锁,抢占到该互斥锁的 HMaster 即为当前 HBase 主服务。当各 slave(HRegionServer)需要知道主服务是谁时,其访问 ZooKeeper 即可。

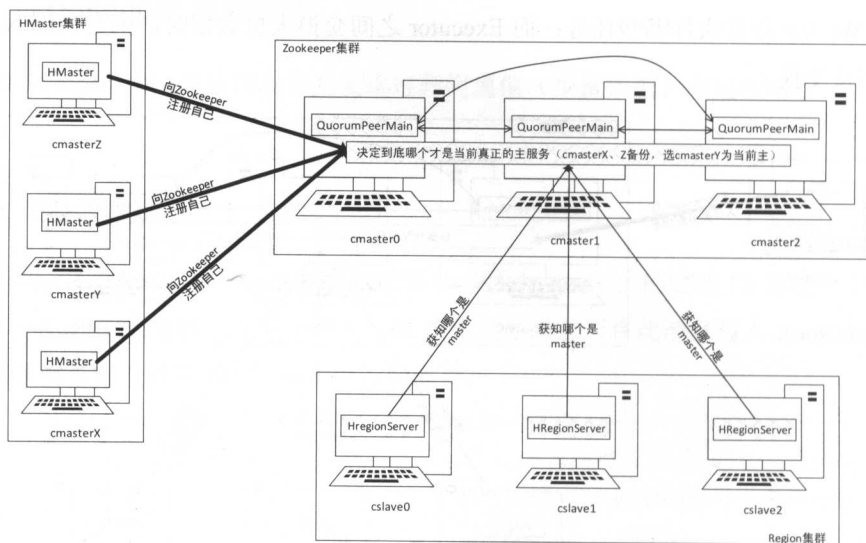


图 7-3 借助 ZooKeeper 实现进程同步

实际上各 slave 每次需要访问 HMaster 时,都会预先访问 ZooKeeper,这是由于真实的 HMaster 可能会发生改变,以图 7-3 为例,假定当前的 cmasterY 宕机,ZooKeeper 会检测到 cmasterY 已不再刷新互斥锁,此时 ZooKeeper 会选中 cmasterZ(也可以是 cmasterX)上的 HMaster 作为新的 HBase 主服务。

7.1.2 ZooKeeper 体系架构

ZooKeeper 部署与执行时，其表现出的特性^[3]并不相同，下面分别讲述。

1. ZooKeeper 部署时

ZooKeeper 的物理拓扑非常简单，既可以只在单机上安装一个 ZooKeeper 服务，也可以在集群上安装 ZooKeeper 集群。单机模式下，该 ZooKeeper 提供一切协调服务；集群模式下，没有 master，slave 之分，统一都是 QuorumPeerMain 进程（图 7-4）。

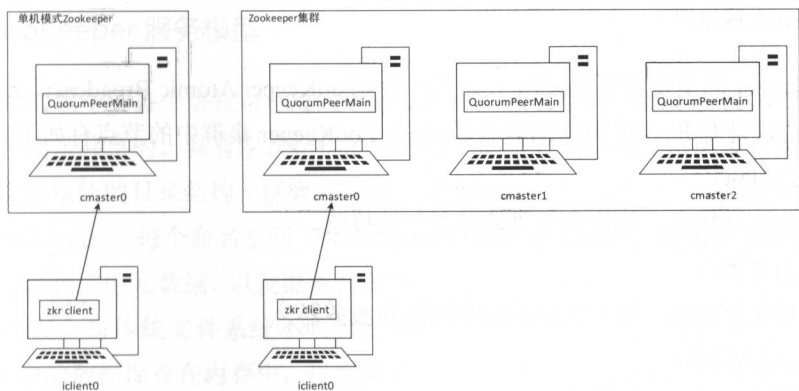


图 7-4 ZooKeeper 部署状态图

2. ZooKeeper 运行时

集群中各台机器上的 ZooKeeper 服务启动后，它们首先会从中选择一个作为领导者，其他则作为追随者，如图 7-5 所示，当只在一台机器上部署 ZooKeeper 时，该 ZooKeeper 就是 Leader；当以集群方式部署 ZooKeeper 时，比如图中 cmaster0、cmaster1 与 cmaster2 上的 ZooKeeper 服务启动后，三者会采取投票方式，以少数服从原则从中选出一个领导者。当发生客户端读写操作时，规定读操作可以在各个节点上实现，不过写操作则必须经领导者同意后方可执行。

ZooKeeper 集群内选取领导时，内部采用的是原子广播协议，此协议是对 Paxos 算法的修改与实现。集群内各个 ZooKeeper 服务选举领导的核心思想是：由某个新加入的服务器发起一次选举，如果该服务器获得 $n/2+1$ 个票数，则此服务器将成为整个 ZooKeeper 集群的领导者。当“领导者”服务器发生故障时，剩下的“追随者”将重新进行新一轮“领导者”选举。因此，集群中 ZooKeeper 个数必须以奇数出现（3、5、7、9...），并且当构建 ZooKeeper 集群时，最少需 3 个节点。

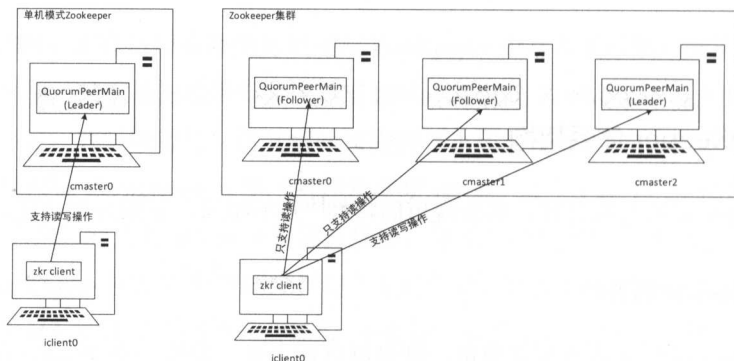


图 7-5 ZooKeeper 运行时状态图

3. Paxos 算法^[4]

ZooKeeper 的实现主要是采用了原子广播（ZooKeeper Atomic Broadcast, Zab）协议。原子广播协议是对 Paxos^[1]算法的修改和补充。ZooKeeper 集群中的节点有如下三种状态：

- LOOKING

表示初始化状态，等待参与“领导者”的投票。

- LEADING

表示领导者状态，统一管理系统中其他的服务器。

- FOLLOWING

表示跟随者状态，服务器中“领导者”被选举成功后，除“领导者”外，剩余的服务器都处于这个状态。

在实际过程中 ZooKeeper 的“领导者”选举有如下两种实现方式^[2]：LeaderElection 和 FastLeaderElection。

（1）LeaderElection

使用 LeaderElection 实现“领导者选举”时，每个服务器会开启一个回复（Response）线程和选举线程。当新增一个服务器时，LeaderElection 会发动一次选举，此时 ZooKeeper 中的每个服务器都会获得当前服务器编号最大的那一台服务器的编号。如果当次编号最大的服务器没有获得 $n/2+1$ 个票数，则重新选举，直到系统中成功的选举出“领导者”。

（2）FastLeaderElection

使用 FastLeaderElection 实现“领导者选举”时，每个服务器都会产生含有三个线程的接收线程池和含有三个线程的发送线程池。在没有选举时，这两个线程池均处于阻塞状态，当新增一个服务器时，FastLeaderElection 会发动一次选举。此时选举线程发起相关的流程操作，通过将自己的编号和用来存储描述“领导者”是否发生变化的变量值通知其他服务器。最后每个服务器都会获得编号最大的服务器的相关信息，在下一次投票时将票投给编号最大的服务器，重复选举过程，直到系统中成功选举出“领导者”。

如果“领导者”失去了响应，所有的“跟随者”都会向“领导者”发送 Ping 消息，如果没有得到响应，那么将会发起新一轮的“领导者”选举。

4. 原子广播

客户端所有的写请求都被转发给“领导者”，“领导者”将收到的请求通过广播的形式发送给所有的“跟随者”，当超过半数的“跟随者”修改数据并持久化后，“领导者”才会提交这个更新，这个过程的执行要么全部成功，要么全部失败。这类似于数据库中的事务提交的过程。

7.1.3 ZooKeeper 服务模型

分布式应用中的各个进程可以通过 ZooKeeper 的命名空间(Namespace)来进行协调，这个命名空间是共享的、具有层次结构的，更重要的是它的结构足够简单，像我们平时接触到的文件系统的目录结构一样容易理解（图 7-6）。

在 ZooKeeper 中每个命名空间（Namespace）被称为 ZNode，而每个 ZNode 包含一个路径和与之相关的元数据，以及继承自该节点的孩子列表。与传统文件系统不同的是，ZooKeeper 中的数据保存在内存中，以实现了分布式同步服务的高吞吐和低延迟。图中 ZooKeeper 的数据模型中，有如下要点。

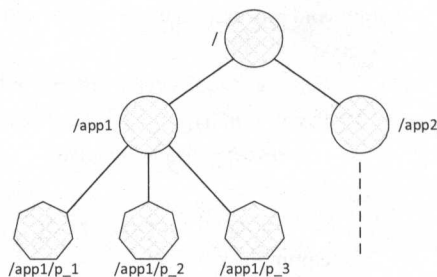


图 7-6 ZooKeeper 树型存储空间

①每个节点（ZNode）中存储的是同步相关的数据（这是 ZooKeeper 设计的初衷，数据量很小，大概 B 到 KB 量级），例如状态信息、配置内容、位置信息等。

②一个 ZNode 维护了一个状态结构，该结构包括：版本号、ACL 变更、时间戳。每次 ZNode 数据发生变化，版本号都会递增，这样客户端的读请求可以基于版本号来检索状态相关数据。

③每个 ZNode 都有一个 ACL，用来限制是否可以访问该 ZNode。

④在一个命名空间中，对 ZNode 上存储的数据执行读和写请求操作都是原子的。

⑤客户端可以在一个 ZNode 上设置一个监视器（Watch），如果该 ZNode 数据发生变更，ZooKeeper 会通知客户端，从而触发监视器中实现的逻辑的执行。

⑥每个客户端与 ZooKeeper 连接，便建立了一次会话（Session），会话过程中，可能发生 CONNECTING、CONNECTED 和 CLOSED 三种状态。

⑦ZooKeeper 支持临时节点（EphemeralNodes）的概念，它是与 ZooKeeper 中的会话

（Session）相关的，如果连接断开，则该节点被删除。

上述特性中提到的会话和版本号在 ZooKeeper 中非常重要，现着重描述如下。

1. 会话

当客户端成功连接到 ZooKeeper 服务器时，与服务器建立了一个会话（Session）。每个会话都存在一个有效时间。如果服务器在有效时间范围内没有收到任何的请求，那么这个会话便会过期。一旦会话过期，便意味着这个会话将再也无法重新打开。已经过期的会话所创建的短暂性的节点也将被删除。

在真实的系统环境中，可以通过 Ping 请求保持会话不过期。

2. 版本号

版本号（Version）是一种乐观加锁的机制，使客户端能够检测出对节点的修改冲突。

3. 监控

当节点的状态发生变化时，监控（Watcher）机制可以让客户端得到通知。

要实现监控的类必须实现 `org.apache.zookeeper.Watcher` 的接口。如下所示：

```
public void process(WatchedEvent event) {
    try {
        Stat stat = zooKeeper.exists(nodePath, false);
        if (stat != null) {
            zooKeeper.delete(nodePath, -1);
        }
    } catch (KeeperException e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

“process”方法是 `org.apache.zookeeper.Watcher` 的接口中定义的方法，当监控条件满足时，此方法被自动调用。在这个例子中通过 `exists` 方法获取传入的节点是否存在，如果存在则先删除它，然后让用户重新创建，以达到修改节点的目的。

7.1.4 ZooKeeper 部署

要取得 ZooKeeper 服务，须首先部署 ZooKeeper，部署时可以采用手工方式部署，也可以使用 Ambari。

1. 手工部署

由于 ZooKeeper 采用对等结构，故在部署 ZooKeeper 时无须指定主从，只在配置文件里写明各 ZooKeeper 所在机地址即可，此外，由于 ZooKeeper 也要使用本地文件系统来存储数据，故还要指定 ZooKeeper 的本地存储目录，下面是在 cmaster0、cmaster1、cmaster2 这三台机器上部署 ZooKeeper 的详细过程。

Step1 在 cmaster0 上，以 allen 身份下载并解压 ZooKeeper。

请读者使用 wget，FireFox 等工具自行下载并使用下述命令解压 ZooKeeper：

```
[allen@cmaster0 ~]$ tar -zxvf ~/zookeeper-3.4.6.tar.gz
```

Step2 将 ZooKeeper 默认配置文件 conf/zoo_sample.cfg 更名为 conf/zoo.cfg。

Step3 配置数据存储空间。

配置文件 zoo.cfg 里默认的存储目录为“dataDir=/tmp/zookeeper”，由于机器重启后，系统会自动清空“/tmp”目录下文件，故须将此目录更改为某固定目录，比如将其配置为“dataDir=/home/allen/Cloud/zkp”。

Step4 配置 ZooKeeper 集群地址。

ZooKeeper 集群可以是一个节点，也可以是多个（奇数），此处配置的 ZooKeeper 集群为三个节点，将下述三行追加到配置文件 zoo.cfg 里即可。

```
server.1=cmaster0:2888:3888
server.2=cmaster1:2888:3888
server.3=cmaster2:2888:3888
```

Step5 将配置好的 ZooKeeper 文件远程拷贝至 cmaster1 和 cmaster2。

```
[allen@cmaster0 ~]$ scp -r zookeeper-3.4.6 cmaster1:~/
[allen@cmaster0 ~]$ scp -r zookeeper-3.4.6 cmaster2:~/
```

Step6 新建并填写各机 ID。

在配置的 ZooKeeper 数据存储目录（当前为“dataDir=/home/allen/Cloud/zkp”）中，新建文件 myid。三台机器按 cmaster0、cmaster1 和 cmaster2 先后顺序分别写入“1”、“2”和“3”。

```
[allen@cmaster0 ~]$ cat /home/allen/Cloud/zkp/myid          #cmaster0 机
1
[allen@cmaster1 ~]$ cat /home/allen/Cloud/zkp/myid          #cmaster1 机
2
[allen@cmaster2 ~]$ cat /home/allen/Cloud/zkp/myid          #cmaster2 机
3
```

Step7 确定存在数据存储目录和 myid 文件。

由于编者配置的 ZooKeeper 数据存储目录为“dataDir=/home/allen/Cloud/zkp”，所以在启动 ZooKeeper 之前，请读者确定 cmaster0、cmaster1、cmaster2 这三台机已存在目录“/home/allen/Cloud/zkp”和文件“/home/allen/Cloud/zkp/myid”。

Step8 启动 ZooKeeper 集群。

使用下述命令启动 ZooKeeper:

```
[allen@cmaster0 ~]$ zookeeper-3.4.6/bin/zkServer.sh start
[allen@cmaster1 ~]$ zookeeper-3.4.6/bin/zkServer.sh start
[allen@cmaster2 ~]$ zookeeper-3.4.6/bin/zkServer.sh start
```

显然, 从 ZooKeeper 的部署和启动可以看出, ZooKeeper 采用对等结构, 不过实际上, ZooKeeper 内部还是领导和被领导关系, 这三个进程启动后, 它们之间会自动选举出一个“领导”, 其他两个 ZooKeeper 则成为追随者, 选举的原则很简单, 就是少数服从多数原则, 这就是为何在 ZooKeeper 集群中, 节点数总是奇数的原因。

Step9 查看 ZooKeeper 是否部署成功。

```
$ netstat -an|grep 3888          #cmaster0、cmaster1、cmaster2 上执行
$ netstat -an|grep 2888          #cmaster0、cmaster1、cmaster2 上执行
```

执行此命令后, 用户可以看到, 各 ZooKeeper 间正在使用此端口通信(选举领导等), 用户还可以使用 jps 命令查看 ZooKeeper 服务进程, 即:

```
$ /usr/java/jdk1.7.0_45/bin/jps -l          #cmaster0、cmaster1、cmaster2 上执行
```

用户能看到 org.apache.zookeeper.server.quorum.QuorumPeerMain, 表示 ZooKeeper 服务已经启动。

虽然在 ZooKeeper 集群内, 各个 ZooKeeper 有“领导者”和“追随者”之分, 但在部署时没有 master/slave 之分, 即在部署和使用, 可以将各台机器的 ZooKeeper 服务看成对等实体, 直接部署与使用即可, 无须关心 ZooKeeper 集群内部如何选举领导、谁是领导。

2. Ambari 部署

使用 Ambari 部署 ZooKeeper 可以说是一键操作, 难点几乎都在 Ambari 工具本身部署上, 以下步骤从无到有, 简单介绍了 Ambari 自身部署和使用 Ambari 部署 ZooKeeper 的大概步骤:

Step1 制定部署规划。

Step2 准备硬件机器和 OS 环境。

Step3 配置单机 OS 环境和集群环境。

Step4 部署 ambari-server。

Step5 使用 ambari-server 部署 ZooKeeper。

例 1 请使用 Ambari 为 littleCstor 部署 ZooKeeper。

解 由于大数据平台涉及太多组件, 故部署之前最好制定一个完备的部署计划, 否则极有可能导致由于各机角色分配混乱而部署失败。

本题以第 3 章为前提, 只给出 ZooKeeper 部署规划表(表 7-1)和部署效果图(图 7-7), 具体部署过程请参见第 3 章。读者须注意, 图 7-7 中 iclient0 到 cmaster0~2 的链接

(图中 Ø 表示) 实际上并不存在, 也就是 iclient0 并不需要向任何机器汇报心跳包, 只有当 iclient0 需要 ZooKeeper 服务时, 它才会主动连接 ZooKeeper。

表 7-1 littleCstor 上 ZooKeeper 部署规划

机 器	角 色	部署服务
cmaster0	ZooKeeper 服务体	zkServer
cmaster1	ZooKeeper 服务体	zkServer
cmaster2	ZooKeeper 服务体	zkServer
iclient0	Client	ZooKeeper Shell 命令行

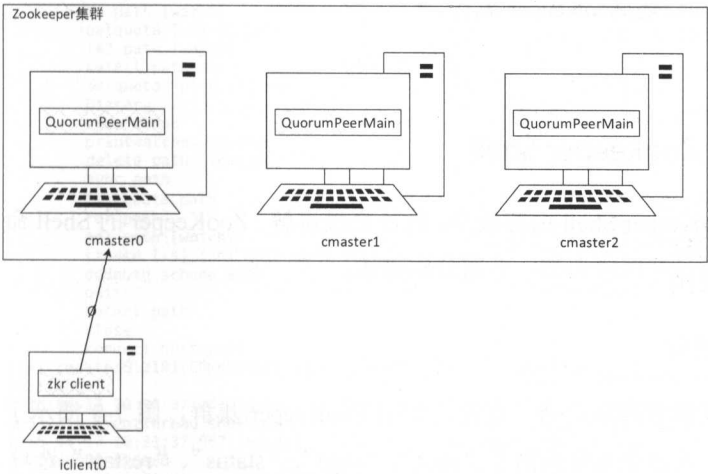


图 7-7 littleCstor 中 ZooKeeper 部署效果图

7.2 ZooKeeper 接口

ZooKeeper 接口指的是用户取得 ZooKeeper 服务的途径, 下面先讲述常见的 ZooKeeper 接口, 接着直接讲述 Shell 接口。

7.2.1 接口汇总

ZooKeeper 的核心作用是提供分布式锁服务, 为提供该功能, 针对不同的上层应用, ZooKeeper 主要提供了 Shell 和 API 访问接口, 即:

- ZooKeeper Shell 接口
- ZooKeeper API 接口

和其他组件不同，ZooKeeper 并没有 Web 接口，这主要是为性能考虑，实际上，用户程序一般都通过读写 ZooKeeper 命名空间来实现进程间同步与互斥，并不需要 Web 页面。

Shell 接口主要为管理员提供，通过该接口，管理员可以开启或关闭 ZooKeeper，新建或查看存储节点等操作。

ZooKeeper API 面向上层应用程序，Java 程序员可通过该接口，实现两个进程间同步操作，7.3 节即讲述 ZooKeeper 编程。

7.2.2 实战 ZooKeeper Shell

由于 ZooKeeper Shell 内容较少，此处直接讲解。ZooKeeper 的 Shell 命令可分为集群管理和命令空间管理两大类，下面分别讲述。

1. 集群管理

所谓的集群管理即启动、查看、关闭 ZooKeeper 集群，图 7-8 演示了该命令，从图中可以看出，该命令主要提供了“start”、“stop”、“status”、“restart”选项。

```
[allen@cmaster0 zookeeper-3.4.6]$ bin/zkServer.sh
JMX enabled by default
Using config: /home/allen/zookeeper-3.4.6/bin/../conf/zoo.cfg
Usage: bin/zkServer.sh {start|start-foreground|stop|restart|status|upgrade|print-cmd}
[allen@cmaster0 zookeeper-3.4.6]$ bin/zkServer.sh status
JMX enabled by default
Using config: /home/allen/zookeeper-3.4.6/bin/../conf/zoo.cfg
Mode: standalone
[allen@cmaster0 zookeeper-3.4.6]$ ssh cmaster2
Last login: Sun Apr 17 09:51:26 2016 from cmaster0
[allen@cmaster2 ~]$ zookeeper-3.4.6/bin/zkServer.sh status
JMX enabled by default
Using config: /home/allen/zookeeper-3.4.6/bin/../conf/zoo.cfg
Mode: leader
[allen@cmaster2 ~]$
```

集群管理命令入口

支持的命令选项

status: 查看本zookeeper状态

已启动，处于“追随者”角色 (★★: cmaster0)

★★: 注意现在在cmaster0，准备登陆cmaster2

★★: 已经登陆到cmaster2 查看cmaster2上zookeeper状态

★★: cmaster2上的Zookeeper处于“领导者”角色

图 7-8 集群管理命令统一入口及其实例

特别地，用户可以使用下述命令，启动 ZooKeeper 集群：

```
[allen@cmaster0 zookeeper-3.4.6]$ bin/zkServer.sh start
[allen@cmaster1 zookeeper-3.4.6]$ bin/zkServer.sh start
[allen@cmaster2 zookeeper-3.4.6]$ bin/zkServer.sh start
```

注意上述命令是在三台不同的机器上分别执行的，当需要关闭集群时，将上述的“start”换成“stop”即可。

2. 命名空间管理

命名空间管理指的是新建、查看或删除节点。图 7-9 为该命令统一入口，进入命令行后，可使用“help”选项，即可查看当前支持的所有操作。

```
[allen@iclient0 zookeeper-3.4.6]$ bin/zkCli.sh -server cmaster0:2181
Connecting to cmaster0:2181
WATCHER::
WatchedEvent state:SyncConnected type:None path:null

[zk: cmaster0:2181(CONNECTED) 0] help
ZooKeeper -server host:port cmd args
  stat path [watch]
  set path data [version]
  ls path [watch]
  delquota [-n|-b] path
  ls2 path [watch]
  setAcl path acl
  setquota -n|-b val path
  history
  redo cmdno
  printwatches on|off
  delete path [version]
  sync path
  listquota path
  rmr path
  get path [watch]
  create [-s] [-e] path data acl
  addauth scheme auth
  quit
  getAcl path
  close
  connect host:port
[zk: cmaster0:2181(CONNECTED) 1] quit
Quitting...
2016-04-17 10:31:37,067 [myid:] - INFO [main-EventThread:ClientCnxn$Ever
d@512] - EventThread shut down
2016-04-17 10:31:37,067 [myid:] - INFO [main:ZooKeeper@684] - Session: 6
23274d0000 closed
[allen@iclient0 zookeeper-3.4.6]$
```

★★：当前在iclient0上

★★：中间省略大量日志

★★：虽然cmaster0为“追随者”连接时，无区别

Zookeeper命名空间管理统一入口

查看所有支持操作

退出Zookeeper命令行

图 7-9 命名空间管理统一入口及其实例

下面的操作即使用上述命令，在 ZooKeeper 存储树中新建一节点并存入信息：

```
[allen@iclient0 zookeeper-3.4.6]$ bin/zkCli.sh -server cmaster0:2181 #iclient0 上,进入 zkr 命令行

[zk: cmaster0:2181(CONNECTED) 0] ls / #查看当前 ZooKeeper 目录结构
[zk: cmaster0:2181(CONNECTED) 1] create /cstorShell chinaCstorShell
[zk: cmaster0:2181(CONNECTED) 2] ls / #查看当前 ZooKeeper 目录结构
[zk: cmaster0:2181(CONNECTED) 3] ls /cstorShell #查看 cstorShell 节点目录结构
[zk: cmaster0:2181(CONNECTED) 4] get /cstorShell #获取 cstorShell 节点信息
[zk: cmaster0:2181(CONNECTED) 5] rmr /cstorShell #删除 cstorShell 节点
[zk: cmaster0:2181(CONNECTED) 6] ls /
[zk: cmaster0:2181(CONNECTED) 7] help #查看所有命令及其帮助
[zk: cmaster0:2181(CONNECTED) 8] quit #退出 ZooKeeper 命令行接口
```

其中“create...”一句含义为“创建节点 cstorShell，并赋予此节点信息 chinaCstorShell”，上述程序执行过程如图 7-10 所示。

```

[zk: cmaster0:2181(CONNECTED) 0] ls /
[zk: cmaster0:2181(CONNECTED) 1] create /cstorShell chinaCstorShell
[zk: cmaster0:2181(CONNECTED) 2] ls /
[zk: cmaster0:2181(CONNECTED) 3] ls /cstorShell
[zk: cmaster0:2181(CONNECTED) 4] get /cstorShell
chinaCstorShell
cZxid = 0xa
ctime = Sun Apr 17 10:48:41 PDT 2016
mZxid = 0xa
mtime = Sun Apr 17 10:48:41 PDT 2016
pZxid = 0xa
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 15
numChildren = 0
[zk: cmaster0:2181(CONNECTED) 5] rmr /cstorShell
[zk: cmaster0:2181(CONNECTED) 6] ls /
[zk: cmaster0:2181(CONNECTED) 7] quit
Quitting...
2016-04-17 10:49:43,505 [myid:] - INFO [main:ZooKeeper@684] - Sess
23274d0002 closed
2016-04-17 10:49:43,507 [myid:] - INFO [main-EventThread:ClientCnx
d@512] - EventThread shut down
[allen@iclient0 zookeeper-3.4.6]$

```

查看有无节点

创建节点

获取节点信息

删除节点

退出Zookeeper

图 7-10 创建节点实例

7.3 实战 ZooKeeper 编程

显然, 使用 Shell 接口管理 ZooKeeper 非常简单, 不过 ZooKeeper 的 API 接口则更加灵活, 比如使用 ZooKeeper 实现上文所述的两进程 Pa 与 Pb 通信等。

ZooKeeper 中提供了一组基于 Java 的 API, 通过 `org.apache.zookeeper.ZooKeeper` 类创建一个实例对象, 连接到 ZooKeeper 服务器然后调用这个类所提供的接口来和服务器进行交互。

实例化 ZooKeeper 对象的语句 “`ZooKeeper zooKeeper = new ZooKeeper(url, session-Timeout, watcher);`” 中各个字段的含义如下。

- url: ZooKeeper 服务器的地址。
- sessionTimeout: 会话持续时间。
- watcher: 监视器, 用于触发相应事件。

1. 创建节点

创建节点的过程一般被使用于向 ZooKeeper 集群系统提交一个新的配置, 或者向该系统添加需要保存的项。节点被创建时需要指明节点被创建的类型 (CreateMode), 具体见表 7-2。

表 7-2 节点创建类型

CreateMode 类型	详解
EPHEMERAL	该节点将在客户端断开连接后删除
EPHEMERAL_SEQUENTIAL	该节点将在客户端断开连接后删除并将其名下附加一个单调递增数
PERSISTENT	该节点在客户端断开连接后不会删除
PERSISTENT_SEQUENTIAL	该节点在客户端断开连接后不会删除,并将其名下附加一个单调递增数

创建节点的示例代码如下:

```
ZooKeeper zooKeeper = new ZooKeeper(url, sessionTimeout, watcher);
zooKeeper.create("/root", "Hello!ZooKeeper".getBytes(),
Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
```

2. 删除节点

当不再使用系统的某个配置节点, 或者某个节点已经失效时, 使用 “delete” 方法删除该节点。例如删除 “/root” 节点, 示例代码如下:

```
ZooKeeper zooKeeper = new ZooKeeper(url, sessionTimeout, watcher);
zooKeeper.delete("/root", -1)
```

3. 获取节点中的存储内容

在 ZooKeeper 已有的节点中, 如果保存了数据, 可以通过 “getData” 方法来获取该节点中的数据, 示例代码如下:

```
ZooKeeper zooKeeper = new ZooKeeper(url, sessionTimeout, watcher);
String value=zooKeeper.getData("/root", false, null);
```

4. 加入子节点

在 ZooKeeper 已有的节点下, 用 “create” 方法指明详细路径, 添加新的节点。

```
ZooKeeper zooKeeper = new ZooKeeper(url, sessionTimeout, watcher);
zooKeeper.create("/root", "Hello!ZooKeeper".getBytes(),
Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
zooKeeper.create("/root/child", "Hello!ZooKeeper".getBytes(),
Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
```

5. 判断节点是否存在

一般使用于初始化操作, 判断某些系统中的配置是否存在。

```
ZooKeeper zooKeeper = new ZooKeeper(url, sessionTimeout, watcher);
Stat stat = zooKeeper.exists("/root", false);
if(stat==null){
System.out.println("此节点不存在");
}else{
```

```
System.out.println("此节点存在");
}
```

7.4 实战 ZooKeeper 之进程通信

对本章开始时提出的问题, 现假设机器 iclient0 上有进程 Pa, 机器 iclient1 上有进程 Pb, 则使用 ZooKeeper 实现进程 Pa 与 Pb 通信时, 可操作如下。

Step1 iclient0 上执行 Pa。

在 iclient0 上编写并执行如下代码, 该代码主要实现在根目录下新建节点 cstorJava, 并存入信息 chinaCstorJava:

```
public class Pa implements Watcher {
    private static final int SESSION_TIMEOUT=5000;    //连接超时时间
    private ZooKeeper zk;                            //ZooKeeper 实例
    private CountDownLatch connectedSignal=new CountDownLatch(1); //同步辅助线程类
    public void connect(String hosts)throws IOException,InterruptedException{//连接 ZooKeeper
        zk=new ZooKeeper(hosts, SESSION_TIMEOUT, this);
        connectedSignal.await();}
    public void process(WatchedEvent event) {
        if (event.getState()==KeeperState.SyncConnected) {
            connectedSignal.countDown();}}
    public void create(String groupName)throws KeeperException,InterruptedException {
        String path="/" + groupName;
        String creatp;
        creatp=zk.create(path,"chinaCstorJava".getBytes(),Ids.OPEN_ACL_UNSAFE,CreateMode.PERSISTENT);
        System.out.println("Created "+createdPath);}
    public void close()throws InterruptedException {zk.close();}
    public static void main(String[] args) throws Exception {
        Pa pa=new Pa();
        pa.connect("cmaster0");
        pa.create("cstorJava");
        pa.close();}}
```

假定此程序打包好后名为 Pa.jar, 存放于/home/allen 目录下, 包名为 njupt.zkp, 则可使用下述命令执行该代码:

```
[allen@iclient0 ~]# java -cp /home/allen/Pa.jar njupt.zkp.Pa #iclient0 上执行 Pa 进程
```

Step2 iclient1 上执行 Pb。

显然, 上述 iclient0 上的 Pa 执行过程是, Pa 向 ZooKeeper 新建目录 cstorJava, 并存入信息 chinaCstorJava, 此后进程 Pa 结束, 此时可在 iclient1 上启动进程 Pb, 读取 ZooKeeper 目录中 cstorJava 节点及其信息, 然后 Pb 结束。在编写 Pb 类时, 只需要将 Pa 类中的 Pa

换成 Pb，并将 create 方法换成下面的 getData 方法：

```
public void getData(String groupName)throws KeeperException,InterruptedException{
    String path="/" +groupName;
    String data=new String(zk.getData(path, false, null));
    System.out.println("ZNode: "+groupName+"\n"+"Its data: "+data);}
```

使用下述命令，在 iclient1 上执行 Pb 即可：

```
[allen@iclient1 ~]# java -cp /home/allen/Pb.jar njupt.zkp.Pb
```

```
#iclient1 上执行 Pb 进程
```

7.5 实战 ZooKeeper 之进程调度系统

有一个 Hadoop 集群，用户不断地向集群中提交任务（Job），在集群运行的过程中提交的任务可能处于等待、执行、执行成功、执行失败中的某个状态。现在使用 ZooKeeper 对其进行集中管理，让第一次处理失败的任务回调后再次执行，当一个任务多次执行失败后，系统将认定此任务存在错误，停止对此任务的回调并将其保存下来。同时，执行成功的任务也要被保存下来。

7.5.1 设计方案

设定节点“/root”为所有被存储节点的根节点，在此节点下设置“/root/wait”、“/root/processed”、“/root/temp”和“/root/error”4个节点存储不同状态的任务。其中，“/root/wait”节点存储当前系统中处于等待状态和执行状态的节点，“/root/processed”节点存储已经被正确处理完的任务，“/root/temp”节点存储第一次执行失败的任务，“/root/error”存储多次执行失败的任务。每个任务以一个子节点的方式存放在对应状态的目录中，节点名称为提交该任务时产生的任务编号（JobID），存储的值为对应的回调操作的 shell 命令。系统结构如图 7-11 所示。

7.5.2 设计实现

1. 系统结构设计

本系统包括如下两个功能模块。

①配置初始化模块：从配置文件中读取配置信息，并用读取到的信息作为服务器启动的参数，主要由类 InitConfReader.java 组成。

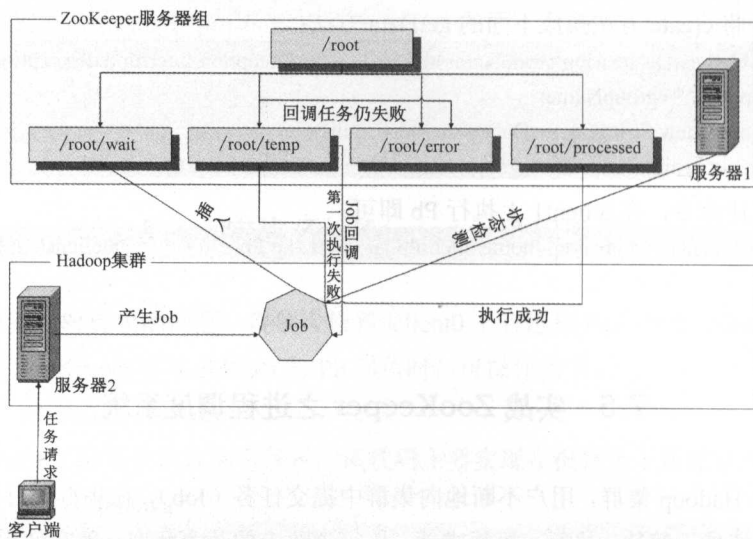


图 7-11 系统结构图

②ZooKeeper 服务器启动、监听模块：不断监听存储在 ZooKeeper 服务器上的任务，当任务执行失败时，进行一次任务的回调，主要由类 SchedulingServer 和 ServerMonitor 组成。

2. 编码

首先在 Eclipse 中新建工程 zookeeperAction，新建包 com.cstore.propertiesReader，com.cstore.zooKeeperServer，导入 ZooKeeper 的核心 Jar 包和 lib 目录下的所有包，导入 Hadoop 的核心包。

在包 com.cstore.propertiesReader 下新建类 InitConfReader.java，负责在系统开启时读取相关配置文件，完成整个系统的配置服务，类 InitConfReader.java 代码如下：

```
public class InitConfReader {
    private String confFileUrl;
    //将文件中存储的配置文件内容存入集合中，以方便开启服务前完成相关配置
    public Map<String, String> getConfs(List<String> keys){
        Map<String, String> result = new HashMap<String, String>();
        Properties properties = new Properties();
        try {
            //将给定位置的配置文件内容读入内存
            properties.load(new FileReader(new File(confFileUrl)));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

        for(String key : keys){
            String value = (String) properties.get(key);
            result.put(key, value);
        }
        return result;
    }
}

```

在包 `com.cstore.zooKeeperServer` 下新建类 `SchedulingServer.java`, 用于初始化和维护系统中的原始节点。系统在初次运行时, 将检测系统中静态节点的状态, 如果静态节点不存在, 则会被重新创建。这里需要考虑所有静态节点中的一个或者多个不存在的情况, 代码如下:

```

public class SchedulingServer implements Watcher {
    private ZooKeeper zooKeeper;
    //connectString 连接字符串, 包括 ZooKeeper 服务器的 IP 地址, ZooKeeper 服务器的端口号
    private String connectString;
    //sessionTimeout 会话超时时间, 单位为毫秒
    private int sessionTimeout;
    //完成服务器开启前, 初始化参数的载入
    public void initConf() throws Exception {
        //初始化需要读取的配置项, 并调用自定义类 InitConfReader 来完成参数加载
        InitConfReader reader = new InitConfReader("init.properties");
        List<String> keys = new ArrayList<String>();
        keys.add("connectString");
        keys.add("sessionTimeout");
        Map<String, String> confs = reader.getConfs(keys);
        this.connectString = confs.get("connectString");
        this.sessionTimeout = Integer.parseInt(confs.get("sessionTimeout"));
        zooKeeper = new ZooKeeper(connectString, sessionTimeout, this);
    }
}

```

//参数初始化配置完成后, 该方法用于完成节点的初始化配置, 创建用于存放不同任务状态的根节点

```

/*
 *系统每次开启时都将检测各个节点的状态, 保证程序可以准确无误的执行
 *在节点检测时, 可能出现多种情况, 必须对当中的每一种情况进行判断
 *1.整个系统中的所有静态节点均未被创建
 *2. “/root” 节点被创建, “/root/client” 未被创建
 *3. “/root/client” 被创建但其下子节点一个或多个未被创建
 *4.存储状态的一个或多个节点未被创建
 */
public void initServer() throws Exception{
    //stat 用于存储被检测节点是否存在, 若不存在则对应的值为 null
    Stat stat = zooKeeper.exists("/root", false);
    if(stat == null){

```

```

//根节点
zooKeeper.create("/root", null, Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
//失败任务存储节点
zooKeeper.create("/root/error", null, Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
//成功任务存储节点
zooKeeper.create("/root/processed", null, Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
//等待和正在运行任务存储节点
zooKeeper.create("/root/wait", null, Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
//临时存储第一次处理失败的节点
zooKeeper.create("/root/temp", null, Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
}
stat = zooKeeper.exists("/root/error", false);
if(stat==null){
    zooKeeper.create("/root/error", null, Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
}
stat = zooKeeper.exists("/root/processed", false);
if(stat==null){
    zooKeeper.create("/root/processed", null, Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
}
stat = zooKeeper.exists("/root/wait", false);
if(stat==null){
    zooKeeper.create("/root/wait", null, Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
}
stat = zooKeeper.exists("/root/temp", false);
if(stat==null){
    zooKeeper.create("/root/temp", null, Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
}
}
public void process(WatchedEvent event) {}
}

```

在包 `com.cstore.zooKeeperServer` 下新建类 `ServerMonitor.java`, 用于监测节点中存储的任务编号 (JobID) 所对应的任务的运行状态, 当状态发生改变时启动对应的操作。

```

public class ServerMonitor implements Watcher, Runnable {
    private ZooKeeper zooKeeper;
    private String connectString;
    private int sessionTimeout;
    private String HadoopHome;
    private String mapredJobTracker;
    //初始化文件加载, 并用其内容配置 ZooKeeper 服务器的连接
    public void initConf() throws Exception {
        InitConfReader reader = new InitConfReader("init.properties");
        List<String> keys = new ArrayList<String>();
        keys.add("connectString");
        keys.add("sessionTimeout");
    }
}

```



```

keys.add("HadoopHome");
keys.add("mapred.job.tracker");
Map<String, String> confs = reader.getConfs(keys);
this.connectString = confs.get("connectString");
this.sessionTimeout = Integer.parseInt(confs.get("sessionTimeout"));
this.HadoopHome = confs.get("HadoopHome");
this.mapredJobTracker = confs.get("mapred.job.tracker");
zooKeeper = new ZooKeeper(connectString, sessionTimeout, this);
}
//监视节点中存储的任务状态变化的
public ServerMonitor() throws Exception {
    SchedulingServer schedulingServer = new SchedulingServer();
    schedulingServer.initConf();
    schedulingServer.initServer();
    initConf();
}
public void process(WatchedEvent event) {}
/*一个任务可能出于：等待，运行，成功，失败，杀死等状态中的一个
*1.任务处于等待和运行状态，不做任何操作，继续监测任务状态，直到状态发生改变
*2.任务处于成功状态，从“/root/client/wait”中删除，并将其插入“/root/client/pr-ocessed”
    当中
*并停止对此节点的状态检测
*3.程序第一次处于失败或杀死状态，将任务插入“/root/client/temp”中，并回调
*如果连续两次都失败或被杀死，则将其插入“/root/client/error”并停止对此
*任务的监测
*/
public void monitorNode() throws Exception {
    List<String> waits = zooKeeper.getChildren("/root/client/wait", false);
    if (!waits.isEmpty()) {
        JobConf conf = new JobConf();
        conf.set("mapred.job.tracker", mapredJobTracker);
        JobClient jobClient = new JobClient(conf);
        for (String wait : waits) {
            String data = new String(zooKeeper.getData("/root/client/wait/"+wait, false, null));
            JobID jobid = null;
            try {
                jobid = JobID.forName(wait);
            } catch (Exception e) {
                System.out.println("job id is wrong!!!");
            }
            Stat stat = zooKeeper.exists("/root/client/error/"+wait, false);
            if (stat!=null){
                zooKeeper.delete("/root/client/error/"+wait, -1);
            }
            ZooKeeper.delete("/root/client/wait/" + wait, -1);

```

```

        ZooKeeper.create("/root/client/error/"+wait, data.getBytes(),
            Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
        continue;
    }
    //通过任务的 JobID 来检测任务正在处于的状态
    int runStat = jobClient.getJob((org.apache.hadoop.mapred.JobID) jobid).getJobState();
    switch (runStat) {
        //处于等待和运行状态的任务在状态不发生改变前不做处理
        case JobStatus.RUNNING:
        case JobStatus.PREP:
            break;
        //当任务执行成功后, 删除原 "/root/wait"
        //目录下的节点并将其任务信息插入 "/root/wait/processed"
        case JobStatus.SUCCEEDED:
            ZooKeeper.delete("/root/client/wait/" + wait, -1);
            ZooKeeper.create("/root/client/processed/"+wait, data.getBytes(),
                Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
            List<String> tempNodes = zooKeeper.getChildren("/root/client/temp", false);
            if(tempNodes == null || tempNodes.size()==0){break;}else{
                for(String tempNode : tempNodes){
                    if(new String(zooKeeper.getData("/root/client/temp/"+tempNode, false, null)).equals(data)){
                        zooKeeper.delete("/root/client/temp/"+tempNode, -1);
                    }
                }
                break;
            }
        //当任务执行失败或者任务被杀掉, 将会把任务插入 "/root/temp" 并回调任务
        //如果回调后任务失败, 则将任务插入 "/root/error"
        case JobStatus.FAILED:
        case JobStatus.KILLED:
            zooKeeper.delete("/root/client/wait/" + wait, -1);
            tempNodes = zooKeeper.getChildren("/root/client/temp", false);
            zooKeeper.create("/root/client/temp/"+wait, data.getBytes(),
                Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
            if(tempNodes == null || tempNodes.size()==0){//用于 shell 命令的回调
                shellTool.callBack(data, HadoopHome);
            }else{boolean flag = true;
                for(String tempNode : tempNodes){
                    if(new String(zooKeeper.getData("/root/client/temp/"+tempNode, false, null)).equals(data)){
                        zooKeeper.create("/root/client/error/"+wait, data.getBytes(),
                            Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
                        zooKeeper.delete("/root/client/temp/" + wait, -1);
                        zooKeeper.delete("/root/client/temp/"+tempNode, -1);
                    }
                }
                flag = false;
            }
    }
    if(flag){ //用于 shell 命令的回调

```

```

        ShellTool.callBack(data, HadoopHome);
    }
    break;
default:
    break;
}
}
}
public void run() {
    try {ServerMonitor serverWaitMonitor = new ServerMonitor();
        while (true) {serverWaitMonitor.monitorNode();
            Thread.sleep(5000);}
        } catch (Exception e) {e.printStackTrace();}}
    public static void main(String[] args) throws Exception {
        Thread thread = new Thread(new ServerMonitor());
        thread.start();
    }
}

```

3. 测试与总结

在一台装有 ZooKeeper 环境的服务器上开启服务，运行 ServerMonitor 类，系统会时刻监测任务的状态变化。当一个新的任务被创建后，该任务的任务编号会被插入 ZooKeeper 中“/root/wait”的目录下，并时刻检测任务状态的改变以便执行不同的处理。

本示例实现了监测系统中任务状态和重新提交失败的任务，在一定程度上避免了整个系统中因为某些特殊原因产生的错误不被处理而造成的不必要损失，大大提高了系统的性能，在实际的应用程序处理过程中具有重要的意义。

7.6 实战 ZooKeeper 之实现 NameNode 自动切换

HDFS 采用主/从式的设计模式。NameNode 上保存了整个 HDFS 的目录树、Block 的位置信息和操作日志，以及这些数据的 Checkpoint。很显然，NameNode 可能存在单点故障，如果 NameNode 失效，整个集群将不可用。Hadoop2.0 之前的版本不支持双 NameNode，2.0 及其以后的 Hadoop 支持双 NameNode。本节讲述借助 Zookeeper 为 Hadoop1.0 配置双 NameNode。注意，该方式非官方源码，可能不够稳定，不过本节意在讲述 Zookeeper 在分布式系统中的协调能力。

7.6.1 设计思想

利用 ZooKeeper 文件系统保存集群需要共享的数据：NameNode 节点和 Standby 节点（备用节点，可能多个）的 IP 地址和端口号。“/namenode”是 Namenode 对应的 znode（在 ZooKeeper 的文件系统中对应一个文件）节点，保存有目前正接管工作的节点 IP 和端口，此节点是临时节点，当持有它的 NameNode 进程关闭之后，这个节点会自动删除。其他备用节点以 Standby 方式启动，这些节点的 IP 和端口信息保存在另一个 znode 节点“/standby”中，备用节点时刻检测代表 NameNode 的“/namenode”节点是否存在，如果“/namenode”不在了，其他的备用节点去争抢创建“/namenode”节点，并将其 IP 和端口号写入“/namenode”中，最终只有一个备用节点抢到这把锁（单一节点的占用可以理解为一把锁）。抢到锁的节点以主节点的方式启动，接管 NameNode 的工作，其他没有抢到锁的节点依然以 standby 的方式运行着，等待下次切换。客户端和 DataNode 可以通过读取 ZooKeeper 中的数据来了解到 NameNode 的变换。主备节点的启动，通过调用脚本来实现。

7.6.2 详细设计

系统可以分为 3 个模块：primarynode 模块、ZooKeeper 模块和脚本执行模块。系统处理的时序图如图 7-12 所示。

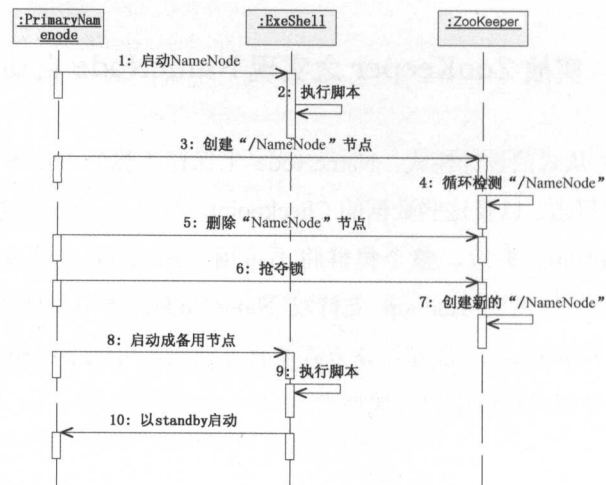


图 7-12 系统时序图

系统包括 5 个类：addMemmber 负责与 ZooKeeper 交互；primarynamenode 代表主备节点；ExeShell 负责调用脚本；tools 提供数据解析的方法；Driver 包含 main 函数，是系统的入口。系统类结构如图 7-13 所示。

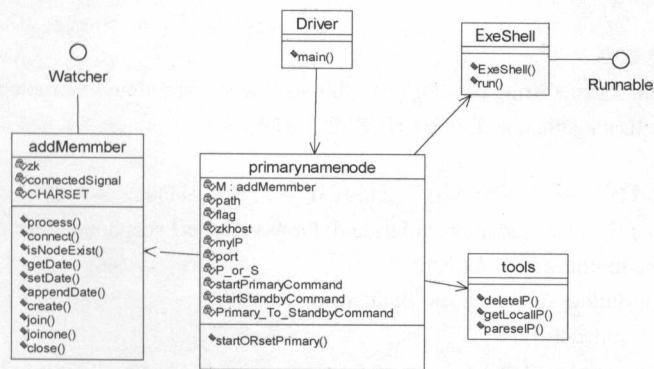


图 7-13 系统类图

7.6.3 编码

新建工程 namenodeSwitch，导入 ZooKeeper 的 jar 包。

新建类 addMemmber.java，实现 Watcher 接口。这个类持有一个 ZooKeeper 实例，负责与 ZooKeeper 集群交互，包括建立连接、判断节点是否存在、读取数据、设置数据、创建节点和关闭连接。代码如下：

```
public class addMemmber implements Watcher{
    private ZooKeeper zk; //zk 实例
    private CountDownLatch connectedSignal=new CountDownLatch(1); //锁计数器
    //编码工具，用于对 zk 中的数据编码
    private static final Charset CHARSET=Charset.forName("UTF-8");
    //1、建立与 zk 的连接
    public void connect(String hosts) throws IOException, InterruptedException{
        int SESSION_TIMEOUT=5000;
        zk=new ZooKeeper(hosts,SESSION_TIMEOUT,this);
        connectedSignal.await();
    }
    //2、判断某个节点是否存在
    public boolean isNodeExit(String path) throws KeeperException, InterruptedException{
        Stat st=zk.exists(path, this);
        if(st==null)
            return false;
        else
            return true;
    }
}
```

```

    }
    //3、读取数据
    public String getetdate(String path) throws KeeperException, InterruptedException {
        byte[] data=zk.getData(path, this, null);
        return new String(data);
    }
    //4、写入数据
    public void setdate(String path,String d) throws KeeperException, InterruptedException {
        zk.setData(path, d.getBytes(CHARSET), -1);
    }
    //5、追加数据，用于多个备用节点将其 IP 信息存入/standby 节点中
    public void appeddate(String path,String d) throws KeeperException, InterruptedException {
        String m=this.getetdate(path);
        StringBuilder sb=new StringBuilder(m);
        sb=sb.append(d);
        String c=sb.toString();
        zk.setData(path, c.getBytes(CHARSET), -1);
    }
    public void process(WatchedEvent event) {
        if(event.getState()==KeeperState.SyncConnected){
            connectedSignal.countDown();}
    }
    //6、创建节点
    public void joinOne(String m) throws KeeperException, InterruptedException {
        String path=m;
        String createdPath=zk.create(path, "1111".getBytes()/*data*/,
        Ids.OPEN_ACL_UNSAFE,CreateMode.EPHEMERAL);
    }
    //7、列出 zk 中的所有节点
    public void list(String group) throws KeeperException, InterruptedException {
        List<String> children=zk.getChildren("/"+group, this);
        if(children.isEmpty())
            System.out.print(group+"* ");
        else {
            System.out.println(group+"^ ");
            for(String a:children){list(group+"/"+a);
            }}}
    }
}

```

新建 `primarynamenode.java`，这个类代表了所有主备节点，持有 `addMemmmber` 的实例，通过调用 `addMemmmber` 的方法来监控 ZooKeeper 集群的状态。`primarynamenode` 在 3 种不同的状态之间切换：主节点、备用节点和下线节点。启动和切换都通过调用来实现，本例中调用的是 Facebook Avatar 的脚本，可以将其替换为其他脚本。关键代码如下：

```

public class primarynamenode {
    private addMemmmber M;    //addMemmmber 的实例，提供与 ZooKeeper 交互的方法
    private String path;      //znode 节点的路径
    private boolean flag;     //标记 “/namenode ” 节点是否存在
}

```

```

private String zkhost;    //任意一台 zk 服务器的 IP
private String myIP;      //本机 IP
private String port;      //HDFS 的监听端口
private int P_or_S=1;    //标记位, 当节点为主节点时设置为 1, 为备用节点设置为 2
private String startPrimaryCommand;    //启动命名
private String startStandbyCommand;
private String Primary_To_StandbyCommand;
public primarynamenode(String zkhost){    //构造函数中初始化变量
    this.M=new addMemmmber();
    this.path="/namenode";
    this.zkhost=zkhost;
    this.flag=false;
    this.myIP=getIP.getLocalIP();
    this.port="9000";
    //Avatar 的启动命令, 可以替换为其他命令
    this.startPrimaryCommand="hadoop org.apache.hadoop.
    hdfs.server.namenode.AvatarNode -zero";
    this.startStandbyCommand="hadoop org.apache.hadoop.hdfs.
    server.namenode.AvatarNode -one -standby -sync";
    this.Primary_To_StandbyCommand="hadoop org.apache.hadoop.
    hdfs.AvatarShell -setAvatar primary";}
//2、依据判断条件切换启动方式
    public void startORsetPrimary() {
        try {//建立与 zk 的连接
            M.connect(zkhost);
            //检查/namenode 节点是否存在,如果存在, 则以 standby 启动
            flag=M.isNodeExit(path);
            if(flag){
                String nowIP=M.getetdate(path);
                //检查存在的/namenode 中的 IP 是否与自身 IP 相等, 如果不等才以 standby 启动
                //并将自身的 IP 追加到/standby 节点中
                if (!nowIP.equals(myIP))
                {
                    if (M.isNodeExit("/standby"))
                    {
                        String exitIp=M.getetdate("/standby");//read the date from exit znode
                        String allIp=exitIp+","+myIP+"."+port;//append myip and port
                        M.setdate("/standby", allIp);
                    }
                    else
                    {
                        M.joinOne("/standby");//创建节点
                        M.setdate("/standby", myIP+"."+port);
                        //执行启动脚本
                        new Thread(new exeShell(startStandbyCommand)).start();
                        Thread.sleep(5000); //等待一段时间, 等启动完毕
                        P_or_S=2;//标记当前为 standby
                    }
                }
            }
        }
    }

```



```
System.out.println(myIP+":start as standby and waiting for being active primary-----");
}}
//假如/namenode 存在，则循环检测
while(flag){
    flag=M.isNodeExit(path);
    Thread.sleep(2000);//check the namenode every two second
}
//当/namenode 不存在了，首先判断自身的当前状态
//如果尚未启动，则直接以主节点启动
//假如现在是 standby 状态，则执行切换命令，切换为主节点
if(!flag){
//1、P_or_S=1 代表目前的状态是尚未启动
if(P_or_S==1){
    System.out.println("-----the primary namenode is not start,ready to start-----");
    try {
        M.joinOne(path);
        M.setdate(path, myIP+": "+port);
        new Thread(new exeShell(startPrimaryCommand)).start();
        Thread.sleep(5000);}
    catch (Exception e) {e.printStackTrace();
        M.close();
    }
}
}
//2、P_or_S=2 代表目前的状态是 standby
if(P_or_S==2){
    System.out.println("-----the primary namenode is down,ready to set to primary-----");
    try {
        M.joinOne(path);
        M.setdate(path, myIP+": "+port);
        new Thread(new exeShell(Primary_To_StandbyCommand)).start();
        M.setdate("/standby", tools.deleteIP(M.getetdate("/standby"), myIP));
        Thread.sleep(5000);
    }catch (Exception e) {
        e.printStackTrace();
        M.close();
    }
}
}
```

Exshell.java 负责调用系统脚本，构造函数中提供脚本的路径，run 方法执行脚本。这个类实现了 Runnable 接口，是个线程类，在 primarynamenode 中被调用，因为执行脚本时一直要等到脚本执行完毕才能返回，而有些脚本的执行是循环等待的，采用单独的线程则无须等待脚本执行返回。Exshell.java 的关键代码如下。

```
public class exeShell implements Runnable{
private String shellname;
public exeShell(String shellname){ this.shellname=shellname; }
public void Exeshell(){
```

```

try{
//执行脚本
    Process process=Runtime.getRuntime().exec(shellname);
    InputStreamReader ir=new InputStreamReader(process.getInputStream());
    LineNumberReader input=new LineNumberReader(ir);
    String line;
//打印脚本的输出
    while((line=input.readLine())!=null)
        System.out.println(line);
    input.close();
    ir.close();
}
catch(Exception e){ e.printStackTrace();}
public void run() { this.Exeshell();}

```

Tools.java 是一个工具类，提供解析 IP 和端口、删除 znode 中 IP 和获取本地 IP 的静态方法，关键代码如下：

```

public class tools {
public static String deleteIP(String srcIP,String desIP){
    StringTokenizer st=new StringTokenizer(srcIP,"");
    StringBuilder b1=new StringBuilder();
    while (st.hasMoreElements()){
        String b=st.nextToken();
        if (b.equals(desIP)){ b1=b1.append(b+","");}
    }
    int length=b1.length();
    b1.deleteCharAt(length-1);
    return b1.toString();
}
public static String getLocalIP(){
    String ip=null;
    try {
        ip=java.net.InetAddress.getLocalHost().getHostAddress();
    } catch (UnknownHostException e) {
        e.printStackTrace();
    }
    return ip;
}
public static List<String> parseIP(String srcIP){
    StringTokenizer st=new StringTokenizer(srcIP,"");
    List<String> s=new ArrayList<String>();
    StringBuilder b1=new StringBuilder();
    while (st.hasMoreElements()){
        s.add(st.nextToken());
    }
    return s;
}
}

```

}

Driver.java 是系统的驱动类, 提供了 main 函数。在 main 函数中首先实例化一个 primarynamenode 对象, 然后调用 primarynamenode 的 startORsetPrimary 方法, 最后让主进程一直处于等待状态。执行时需要提供一个 ZooKeeper 服务器的 IP 地址来作为传入参数。代码如下。

```
public class Driver {
    public static void main(String[] args) throws InterruptedException{
        primarynamenode pn=new primarynamenode(args[0]);
        pn.startORsetPrimary();
        System.out.println("keeper link to zkhost:"+args[0]);
        Thread.sleep(Long.MAX_VALUE);
    }
}
```

启动 ZooKeeper 集群, 将以上代码打成 jar 包, 分发到所有作为 NameNode 的主备节点上, 依次运行代码, 则第一个运行的节点将成为集群中真正工作的 NameNode, 其他节点都以备用的状态启动。当 NameNode 宕机后, 在备用节点中会产生一个新的 NameNode。

7.6.4 实战总结

本示例可以运用到 Hadoop 的各个版本上, 能够实现 NameNode 的自动切换, 无须人工干预, 为集群的管理提供了便捷。本示例的备用节点只是冷备, 如果来实现热备, 需要对 NameNode 上的数据做其他的处理, 可参考“HDFS High Availability Using the Quorum Journal Manager”, 该项目网址为: <http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html>。

习 题

1. 进程间为什么有同步与互斥? 单机环境下可采用什么机制解决同步与互斥? 集群环境下又如何?
2. 简述 ZooKeeper 框架功能作用及其体系架构。
3. ZooKeeper 内部采用什么算法来选举 Leader? 采用什么算法来确保数据一致性? 为何 ZooKeeper 节点必须奇数个?
4. 简述手工部署 ZooKeeper、使用 Ambari 部署 ZooKeeper 的部署步骤。
5. 简述 ZooKeeper 访问接口。

6. 简述使用 Maven 和不使用 Maven 时, ZooKeeper 开发环境搭建部署。
7. 简述 ZooKeeper 命令空间数据结构以及节点类型。
8. 在大型系统中, 如何使用 ZooKeeper 来实现进程通信、进程协调? 当借助 ZooKeeper 实现进程通信时, 如何避免数据灾难?
9. 简述在 HBase、Hama 和 Storm 中, ZooKeeper 的层次位置和功能作用。在 HDFS-HA 和 YARN-HA 中 ZooKeeper 如何发挥作用?
10. YARN 的资源仲裁模块为什么不用 ZooKeeper 实现?
11. 在大型集群中, 如何确保 ZooKeeper 集群自身安全可靠?
12. 在一个大型集群中, 对于一个长期运行的 ZooKeeper, 如何确保其内存占用紧凑?

参考文献

- [1] https://en.wikipedia.org/wiki/Mutual_exclusion
- [2] <http://zookeeper.apache.org/>
- [3] <http://zookeeper.apache.org/doc/trunk/zookeeperStarted.html>
- [4] Lamport L. Paxos made simple[J]. ACM Sigact News, 2001, 32(4): 18-25.

HBase^[1]是基于 Hadoop 的开源分布式数据库,它以 Google 的 BigTable 为原型,设计并实现了具有高可靠性、高性能、列存储、可伸缩、实时读写的分布式数据库系统。HBase 不仅仅在其设计上不同于一般的关系型数据库,在功能上区别更大,表现在其适合于存储非结构化数据,而且 HBase 是基于列的而不是基于行的模式。就像 BigTable 利用 GFS (Google 文件系统)所提供的分布式存储一样,HBase 在 Hadoop 之上提供了类似于 BigTable 的能力。

8.1 HBase 简介

2006 年谷歌发表论文 BigTable,年末,微软旗下自然语言搜索公司 Powerset 出于处理大数据的需求,按论文思想,开启了 HBase 项目并于 2008 年将其捐赠给 Apache,2010 年 HBase 成为 Apache 顶级项目。下面首先介绍 HBase 体系架构,接着介绍其数据模型和集群部署。

8.1.1 体系架构

显然数据库应具有的核心功能是数据存储和访问,作为分布式数据库,HBase 提供了横向扩展机制,HBase 的体系架构^[2]中,能够充分体现这几大功能。

1. 集群成员

HBase 采用 master/slave 架构,主节点运行的服务称为 HMaster,从节点服务称为 HRegionServer,底层采用 HDFS 存储数据。HMaster 负责管理多个 HRegionServer、恢复 HRegionServer 的故障等。HRegionServer 负责多个区域的管理以及相应客户端请求。HRegionServer 还负责区域划分并通知 HMaster 有了新的子区域(图 8-1)。

HBase 需要 ZooKeeper 集群服务,默认情况下,它管理一个 ZooKeeper 实例,作为“权威机构”。ZooKeeper 会记录 HMaster 位置、根目录表位置等核心数据,此时若有 HRegionServer 崩溃,就可以通过 ZooKeeper 来进行分配协调。此外,当 HBaseClient 连接到 HBase 时,其须首先访问 ZooKeeper,在获取 HMaster、HRegionServer、-ROOT-等核心数据后,方可连接到 HRegionServer。

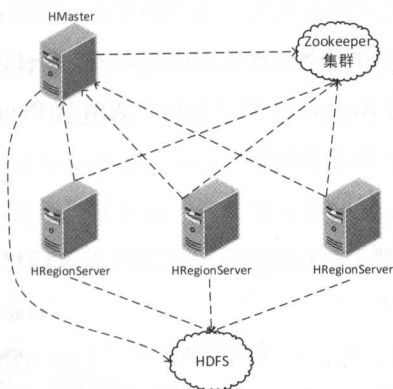


图 8-1 HBase 集群成员

1) Region 服务器

HBase 在行的方向上将表分成了多个 Region，每个 Region 包含了一定范围内（根据行键进行划分）的数据。每个表最初只有一个 Region，随着表中的记录数不断增加直到超过某个阈值时，Region 就会被分割形成两个新的 Region。所以一段时间后，一个表通常会含有多个 Region。Region 是 HBase 中分布式存储和负载均衡的最小单位，即一个表的所有 Region 会分布在不同的 Region 服务器上，但一个 Region 内的数据只会存储在一个服务器上。物理上所有数据都存储在 HDFS 上，并由 Region 服务器来提供数据服务，通常一台计算机只运行一个 Region 服务器程序（HRegionServer），每个 HRegionServer 管理多个 Region 的实例（HRegion），如图 8-2 所示。其中 HLog 是用来做灾难备份的，它使用的是预写式日志（Write-Ahead Log, WAL）。每个 Region 服务器只维护一个 HLog，所以来自不同表的 Region 日志是混合在一起的，这样做的目的是不断追加单个文件，相对于同时写多个文件而言，可以减少磁盘寻址次数，因此可以提高对表的写性能。带来的麻烦是，如果一台 Region 服务器下线，为了恢复其上的 Region，需要将 Region 服务器上的 Log 进行拆分，然后分发到其他 Region 服务器上进行恢复。

每个 Region 由一个或多个 Store 组成，每个 Store 保存一个列族的所有数据。每个 Store 又是由一个 memStore 和零个或多个 StoreFile 组成，StoreFile 则是以 HFile 的格式存储在 HDFS 上的，如图 8-3 所示。

当客户端进行更新操作时，先连接有关的 HRegionServer，然后向 Region 提交变更。提交的数据会首先写入 WAL（Write-Ahead Log）和 MemStore 中，当 MemStore 中的数据累计到某个阈值时，HRegionServer 就会启动一个单独的线程将 MemStore 中的内容刷新到磁盘，形成一个 StoreFile。当 StoreFile 文件的数量增长到一定阈值后，就会将多个 StoreFiles 合并（Compact）成一个 StoreFile，合并过程中会进行版本合并和数据删除，因

此可以看出 HBase 其实只有增加数据, 所有的更新和删除操作都是在后续的合并过程中进行的。StoreFiles 在合并过程中会逐步形成更大的 StoreFile, 当单个 StoreFile 大小超过一定阈值后, 会把当前的 Region 分割 (Split) 成两个 Regions, 并由 HMaster 分配到相应的 Region 服务器上, 实现负载均衡。

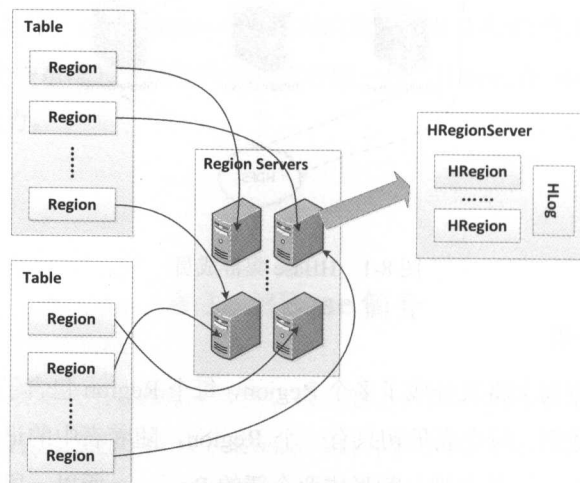


图 8-2 Region 服务器

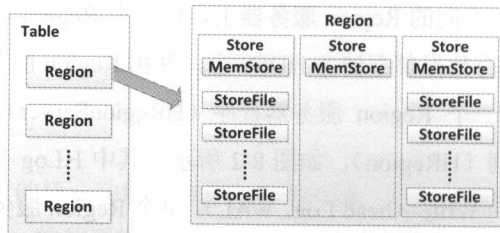


图 8-3 Region 示意图

2) 主服务器

HBase 每个时刻只有一个 HMaster (主服务器程序) 在运行, HMaster 将 Region 分配给 Region 服务器, 协调 Region 服务器的负载并维护集群的状态。HMaster 不会对外 (Region 服务器和客户端) 提供数据服务, 而是由 Region 服务器负责所有 Regions 的读写请求及操作。如果 HRegionServer 发生故障终止后, HMaster 会通过 ZooKeeper 感知到, 并处理相应的 Log 文件, 然后将失效的 Regions 进行重新分配。此外, HMaster 还负责管理表的 schema 和对元数据的操作。

由于 HMaster 只维护表和 Region 的元数据, 而不参与数据的输入输出过程, HMaster 失效仅会导致所有的元数据无法被修改, 但表的数据读写还是可以正常进行的。

3) 元数据表

用户表的 Regions 元数据被存储在.META.表中，随着 Region 的增多，.META.表中的数据也会增大，并分裂成多个 Regions。为了定位.META.表中各个 Regions 的位置，把.META.表中所有 Regions 的元数据保存在-ROOT-表中，最后由 ZooKeeper 记录-ROOT-表的位置信息。所以客户端访问用户数据前，需要首先访问 ZooKeeper 获得-ROOT-的位置，然后访问-ROOT-表获得.META.表的位置，最后根据.META.表中的信息确定用户数据存放的位置，如图 8-4 所示。

-ROOT-表永远不会被分割，它只有一个 Region，这样可以保证最多需要三次跳转就可以定位任意一个 Region。为了加快访问速度，.META 表的 Regions 全都保存在内存中，如果.META.表中的每一行在内存中大约占 1KB，且每个 Region 限制为 128MB，那么图 8-4 所示的三层结构可以保存的 Regions 数目为： $(128\text{MB}/1\text{KB}) * (128/1\text{KB}) = 234$ 个。客户端会将查询过的位置信息缓存起来，且缓存不会主动失效。如果客户端根据缓存信息还访问不到数据，则询问持有相关.META.表的 Region 服务器，试图获取数据的位置，如果还是失效，则询问-ROOT-表相关的.META.表在哪里。最后，如果前面的信息全部失效，则通过 ZooKeeper 重新定位 Region 的信息。所以如果客户端上的缓存全部是失效，则需要进行 6 次网络来回，才能定位到正确的 Region。

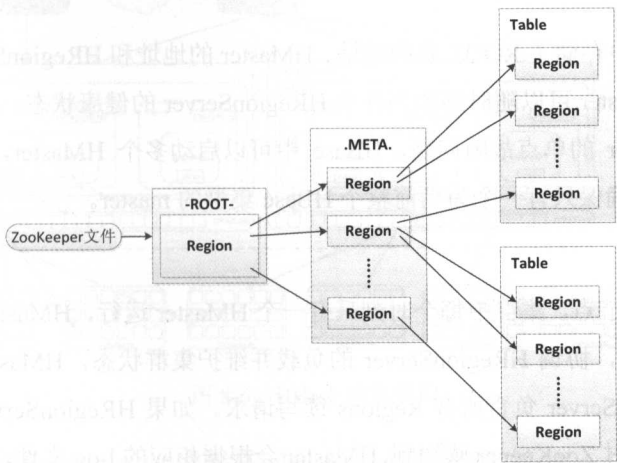


图 8-4 Region 定位示意图

2. 运行时 HBase

图 8-5 是其典型的物理拓扑图，cmaster2 上部署了 HMaster，各 slave 上均部署了 HRegionServer，iclient0 上部署了 Client 接口，底层采用 HDFS 存储数据。

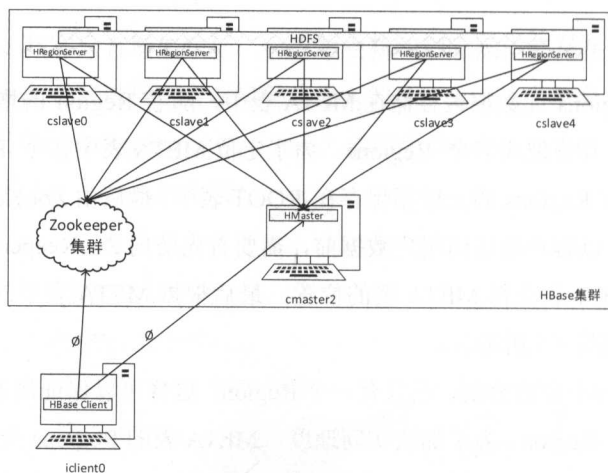


图 8-5 HBase 典型物理拓扑

1) Client

Client 端使用 HBase 的 RPC 机制与 HMaster 和 HRegionServer 进行通信，对于管理类操作，Client 与 HMaster 进行 RPC；对于数据读写类操作，Client 与 HRegionServer 进行 RPC。

2) ZooKeeper

ZooKeeper 中存储了 ROOT 表的地址、HMaster 的地址和 HRegionServer 地址，通过 ZooKeeper，HMaster 可以随时感知到各个 HRegionServer 的健康状态。此外，ZooKeeper 也避免了 HMaster 的单点故障问题，HBase 中可以启动多个 HMaster，通过 ZooKeeper 的选举机制能够确保只有一个为当前整个 HBase 集群的 master。

3) HMaster

即 HBase 主节点，集群中每个时刻只有一个 HMaster 运行，HMaster 将 Region 分配给 HRegionServer，协调 HRegionServer 的负载并维护集群状态，HMaster 对外不提供数据服务，HRegionServer 负责所有 Regions 读写请求。如果 HRegionServer 发生故障终止后，HMaster 会通过 ZooKeeper 感知到，HMaster 会根据相应的 Log 文件，将失效的 Regions 重新分配，此外 HMaster 还管理用户对 Table 的增、删、改、查操作。

4) HRegionServer

HRegionServer 主要负责响应用户 I/O 请求，向 HDFS 文件系统中读写数据，其内部管理了一系列 HRegion 对象，当 StoreFile 大小超过一定阈值后，会触发 Split 操作，即将当前 Region 拆成两个 Region，父 Region 会下线，新 Split 出的两个孩子 Region 会被 HMaster 分配到相应的 HRegionServer 上。

3. 数据存取过程

当 HBase 对外提供服务时，其内部存储着名为 -ROOT- 和 .META. 的特殊目录表，它们维护着当前集群上所有区域的列表、状态和位置信息。-ROOT- 表包含 .META. 表的区域列表，.META. 包含所有用户空间区域列表，表中的项则使用区域名作为键。当区域变化时，目录表会进行相应更新，这样，集群上所有区域信息就能保持最新。

新连接到 ZooKeeper 集群上的客户端首先查找 -ROOT- 的位置，然后客户端通过 -ROOT- 获取请求行所在范围所属 .META. 区域位置，接着，客户端查找 .META. 区域位置来获取用户空间区域所在节点及其位置，最后，客户端即可直接和管理该区域的 HRegionServer 进行交互。

一旦 Client 知道了数据的实际位置（即某 HRegionServer 位置），该 Client 会直接和该 HRegionServer 进行交互，此时 HRegionServer 会打开并创建对应的 HRegion 实例。当 HRegion 被打开后，它会为每个表的 HColumnFamily 创建一个 Store 实例，这些列族是用户之前创建表时定义的。每个 Store 实例包含一个或多个 StoreFile 实例，它们是实际数据存储文件 HFile 的轻量级封装，每个 Store 还持有一个 MemStore，一个 HRegionServer 共享一个 HLog 实例（图 8-6）。

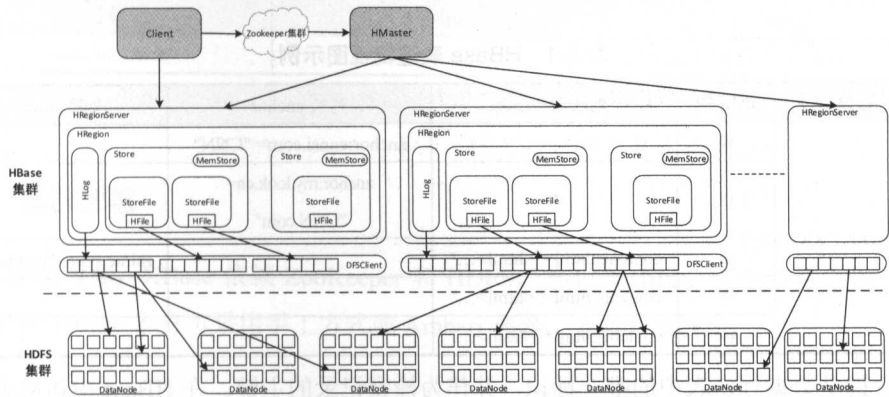


图 8-6 HBase 体系架构

当 Client 向 HRegionServer 发起 HTable.put(put) 请求时，该 HRegionServer 会将请求交给对应的 HRegion 实例来处理。此时，第一步是要决定数据是否需要写到由 HLog 类实现的预写日志（WAL）中，当采用预写机制时，当服务器崩溃时可回滚还没有持久化的数据。

一旦数据被写入 WAL（Write ahead log）中，数据就会被放到 MemStore（内存）中，同时还会检查 MemStore 是否已经写满，如果满了，就会请求刷新到磁盘中。刷新请求由 HRegionServer 的一个新线程处理，它会把数据写成 HDFS 中的一个新 HFile，同时还会

保存最后写入的序号，这样，系统可知道哪些数据被持久化到了 HDFS 中。

8.1.2 数据模型

数据库一般以表的形式存储结构化数据，HBase 也以表的形式存储数据，我们称用户对数据的组织形式为数据的逻辑模型，HBase 里数据在 HDFS 上的具体存储形式则称为数据的物理模型。

1. 逻辑模型

HBase 以表的形式存储数据，每个表由行和列组成，每个列属于一个特定的列族（Column Family）。表中的行和列确定的存储单元称为一个元素（Cell），每个元素保存了同一份数据的多个版本，由时间戳（Time Stamp）来标识。表 8-1 给出了 www.cnn.com 网站的数据存放逻辑视图，表中仅有一行数据，行的唯一标识为 com.cnn.www，对这行数据的每一次逻辑修改都有一个时间戳关联对应。表中共有四列：contents:html，anchor:cnnsi.com，anchor:my.look.ca，mime:type，每一列以前缀的方式给出其所属的列族。

表 8-1 HBase 表逻辑视图示例

行键	时间戳	列族 contents	列族 anchor	列族 mime
"com.cnn.www"	t9		anchor:cnnsi.com= "CNN"	
	t8		anchor:my.look.ca= "CNN.com"	
	t6	contents:html="<html>..."		mime:type="text/html"
	t5	contents:html="<html>..."		
	t6	contents:html="<html>..."		

行键是数据行在表中的唯一标识，并作为检索记录的主键。在 HBase 中访问表中的行只有三种方式：通过单个行键访问、给定行键的范围访问、全表扫描。行键可以是任意字符串，默认按字段顺序进行存储。

表中的列定义为：<family>:<qualifier>（<列族>:<限定符>），如 contents:html。通过列族和限定符两部分可以唯一指定一个数据的存储列。

时间戳对应着每次数据操作所关联的时间，可以由系统自动生成，也可以由用户显示地赋值。如果应用程序需要避免数据版本冲突，则必须显示地生成时间戳。HBase 提供了两个版本的回收方式：一是对每个数据单元，只存储指定个数的最新版本；二是保存最近一段时间内的版本（如七天），客户端可以按需查询。

元素由行键、列（<列族>:<限定符>）和时间戳唯一确定，元素中的数据以字节码的

形式存储，没有类型之分。

2. 物理模型

HBase 是按照列存储的稀疏行/列矩阵，其物理模型实际上就是把概念模型中的一个行进行分割，并按照列族存储，见表 8-2。从表中可以看出表中的空值是不被存储的，所以查询时间戳为 t8 的 contents:html 将返回 null。如果没有指名时间戳，则返回指定列的最新数据值，如不指明时间戳时查询 contents:，将返回 t6 时刻的数据。容易看出，可以随时向表中的任何一个列添加新列，而不需要事先声明。

表 8-2 HBase 表物理存储示例

行键	时间戳	列族 contents
"com.cnn.www"	t6	contents:html="<html>..."
	t5	contents:html="<html>..."
	t3	contents:html="<html>..."
行键	时间戳	列族 anchor
"com.cnn.www"	t9	anchor:cnnsi.com="CNN"
	t8	anchor:my.look.ca="CNN.com"
行键	时间戳	列族 mime
"com.cnn.www"	t6	mime:type="text/html"

8.1.3 集群部署

由前文可知，HBase 依赖 ZooKeeper 和 HDFS，故部署 HBase 前须部署 ZooKeeper 和 HDFS，HBase 本身可采用手工方式或 Ambari 部署，下面分别讲述。

1. 手工方式

假定现有机器 cmaster0、cslave0~3、iclient0 和一个现成的 ZooKeeper 集群，下面以手工方式在这些机器上部署 HBase，步骤如下。

Step1 制定部署规划。

根据机器名，编者做出的规划为：cmaster0 机充当主节点，部署 HMaster 服务；cslaveX 充当从节点，部署从属服务 HRegionServer 进程；iclient0 充当客户节点，部署客户端，图 8-7 为部署规划的效果图。

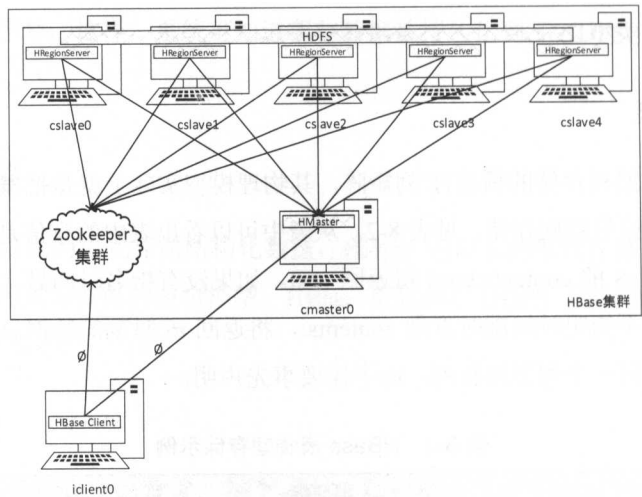


图 8-7 独立模式的 HBase 集群效果图

Step2、3、4、5:

在制定好上述部署规划后，请读者自行准备硬件机器，各台机器的操作系统统一为 CentOS-6.7-x86_64，除了默认的 root 用户外，统一添加 allen 用户，编者将以 allen 用户来安装 HBase。在 HBase 部署之前，需要对集群中每台机器，修改其机器名，添加域名映射，关闭防火墙，安装 jdk，这些操作命令编者已经在第 3 章讲述过，故不再赘述。

Step6 部署 HDFS、ZooKeeper。

Step7 主节点解压 HBase，配置 HBase。

One 下载 HBase，将 HBase 拷至 cmaster0 机。

在下载 HBase 时，请读者到官网下载最新的稳定版 HBase，比如编者下载的就为当前最新稳定版“hbase-1.1.2-bin.tar.tgz”。

Two 在 cmaster0 上，使用如下命令，解压“hbase-1.1.2-bin.tar”。

```
[allen@cmaster0 ~]$ tar -zxvf hbase-1.1.2-bin.tar
```

Three 编辑“hbase-1.1.2/conf/hbase-env.sh”，定位到如下一行：

```
# export JAVA_HOME=/usr/java/jdk1.6.0/
```

将其更改为：

```
export JAVA_HOME=/usr/java/jdk1.8.0_20
```

其中“/usr/java/jdk1.8.0_20”为编者 jdk 安装目录，若读者该目录不相同，请相应修改。

Four 编辑“hbase-1.1.2/conf/hbase-site.xml”文件，将下述内容加入该文件：

```
<property>
  <name>hbase.cluster.distributed</name>
  <value>true</value>
</property>
```

```

<property>
  <name>hbase.rootdir</name>
  <value>hdfs://cmaster0:8020/user/hbase</value>
</property>
<property>
  <name>hbase.zookeeper.quorum</name>
  <value>cmaster0,cmaster1,cmaster2</value>
</property>
<property>
  <name>hbase.zookeeper.property.dataDir</name>
  <value>/home/allen/cloud/zkr</value>
</property>

```

Five 编辑文件“hbase-1.1.2/conf/regionservers”，首先删除该文件已有内容，接着写入 cslave0~3。操作结束后，使用 cat 命令确保内容如下：

```

[allen@cmaster0 conf]$ cat regionservers
cslave0
cslave1
cslave2
cslave3
[allen@cmaster0 conf]$

```

Six 确保 ZooKeeper 集群存在“cloud/zkr”目录。

由于 ZooKeeper 要使用“/home/allen/cloud/zkr”，故须确保部署 ZooKeeper 的三台机器均存在该目录，若不存在，须使用下述命令逐一新建：

```

[allen@cmaster0 ~]$ mkdir -p /home/allen/cloud/zkr
[allen@cmaster1 ~]$ mkdir -p /home/allen/cloud/zkr
[allen@cmaster2 ~]$ mkdir -p /home/allen/cloud/zkr

```

至此 HBase 配置结束。

Step8 将配置好的 HBase 复制至所有 slave 机和 iclient0。

将配置好的 hbase-1.1.2 拷贝至 cslave0~cslave3，iclient0。下面的命令可实现将 hbase-1.1.2 同时拷贝至 3 台 slave。

```

[allen@cmaster0 ~]$ for x in `cat hbase-1.1.2/conf/regionservers`;do echo $x ;scp -r hbase-1.1.2 allen@$x:~/;done;

```

接着使用下述命令将 hbase-1.1.2 拷贝至 iclient0：

```

[allen@cmaster0 ~]$ scp -r hbase-1.1.2 iclient0:~/

```

Step9 启动 HBase 集群。

启动 HBase 的前提是启动 HDFS 和 Zookeeper。此时，若 HBase 自身托管 Zookeeper 集群时，则无需手工启动 Zookeeper，若 Zookeeper 为一独立集群，则用户需首先启动 Zookeeper 集群。由于笔者的 HBase 自动管理 Zookeeper 集群，故只需启动 HDFS，无需手工启动 Zookeeper，启动 HBase 步骤如下：

One 启动 HDFS 集群。

```
[allen@cmaster0 Hadoop-2.7.1]$ sbin/start-dfs.sh
```

Two 启动 HBase。

```
[allen@cmaster0 ~]$ hbase-1.1.2/bin/start-hbase.sh
```

上述命令会自动启动 ZooKeeper 集群，此外，该命令只能在 cmaster0 上执行，不可以其他 slave 机或客户机上执行，脚本默认本机即为 HBase 主节点。和 start 命令类似，用户可以使用 stop-hbase.sh 命令关闭整个 HBase 集群。

Step9 验证 HBase 是否成功启动。

One 验证进程。

在 HBase 集群启动后，用户可以通过在 cmaster0 和 cslave0~3 上分别执行 jps 命令查看到对应进程，即 cmaster0 将显示进程为 HMaster 和 HQuorumPeer（图 8-8），slave 机进程为 HRegionServer（图 8-9）。

```
[allen@cmaster0 ~]$ /usr/java/jdk1.8.0_20/bin/jps
22707 HMaster
29491 Jps
20263 SecondaryNameNode
22647 HQuorumPeer
20061 NameNode
[allen@cmaster0 ~]$
```

图 8-8 cmaster0 进程汇总

图 8-8 展示了 cmaster0 节点上运行 jps 命令的输出结果。输出列表包含：22707 HMaster、29491 Jps、20263 SecondaryNameNode、22647 HQuorumPeer、20061 NameNode。图中有箭头指向 HMaster 和 HQuorumPeer，标注为“HBase 主进程”；指向 SecondaryNameNode 和 NameNode，标注为“Zookeeper 主进程”；指向 NameNode，标注为“HDFS 主进程”。右侧有一个向上箭头指向 jps 命令，标注为“jps 命令，查看 java 进程”。

```
[allen@cslave2 ~]$ /usr/java/jdk1.8.0_20/bin/jps
16192 HRegionServer
21027 Jps
14803 DataNode
[allen@cslave2 ~]$
```

图 8-9 cslave2 进程汇总

图 8-9 展示了 cslave2 节点上运行 jps 命令的输出结果。输出列表包含：16192 HRegionServer、21027 Jps、14803 DataNode。图中有箭头指向 HRegionServer，标注为“HBase 主进程”；指向 DataNode，标注为“HDFS 主进程”。右侧有一个向上箭头指向 jps 命令，标注为“jps 命令，查看 java 进程”。

Two 验证 WebUI。

此外，用户还可在 FireFox 浏览器里输入地址“http://cmaster0:16010”，即可看到 HBase 的 Web UI，此页面上包含了 HBase 集群主节点、从节点等各类统计信息。HRegionServer 的 Web UI 位置为“HRegionServerIP:16030”，读者可任意打开四个 slave 机的 HRegionServer Web UI。

Three 验证新建 HBase 表。

当用户拥有了自己的一个 HBase 集群后，最想做的事应该是新建一个 HBase 表格，下述命令完成以 allen 身份，在 iclient0 上启动 HBase Client 端并新建一表格（图 8-10）：

```
[allen@iclient0 hbase-1.1.2]$ bin/hbase shell
hbase(main):001:0> create 'test', 'cf'
0 row(s) in 3.2080 seconds
=> Hbase::Table - test
hbase(main):002:0>
```



```
[allen@iclient0 hbase-1.1.2]$ bin/hbase shell
hbase(main):001:0> list
TABLE
0 row(s) in 0.4910 seconds

=> []
hbase(main):002:0> create 'test' 'cf'
0 row(s) in 1.2890 seconds

=> Hbase::Table - test
hbase(main):003:0> list
TABLE
test
1 row(s) in 0.0170 seconds

=> ["test"]
hbase(main):004:0>
```

iclient0进入Hbase客户端

显示所有表

新建test表

显示所有表

成功新建了test表

图 8-10 cslave2 进程汇总

在上述命令执行过程中，读者可在 Firefox 浏览器里打开“http://cmaster0:16010”，在该地址中可看到新建的 Table；此外，读者也可打开“http://cmaster0:50070”，接着依次定位“Utilities→Browse the file system”，进入目录“/user/hbase/data/default”，即可查看到刚才新建的 test 表在 HDFS 中的痕迹（图 8-11）。

Browse Directory

HBaSe在HDFS中存储地址

HBaSe在HDFS中存储状态

/user/hbase

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
drwxr-xr-x	allen	supergroup	0 B	2016/1/18 下午11:56:13	0	0 B	.top
drwxr-xr-x	allen	supergroup	0 B	2016/1/18 下午11:42:48	0	0 B	MasterProchfs
drwxr-xr-x	allen	supergroup	0 B	2016/1/18 下午11:42:53	0	0 B	WLS
drwxr-xr-x	allen	supergroup	0 B	2016/1/18 下午11:53:48	0	0 B	archive
drwxr-xr-x	allen	supergroup	0 B	2016/1/18 上午6:28:42	0	0 B	data
-rwxr-xr-x	allen	supergroup	42 B	2016/1/18 上午6:28:33	3	128 MB	hbase.id
-rwxr-xr-x	allen	supergroup	7 B	2016/1/18 上午6:28:32	3	128 MB	hbase.version
drwxr-xr-x	allen	supergroup	0 B	2016/1/18 下午11:51:48	0	0 B	oldfiles

依次进入 data.default 将看到 test 表

图 8-11 HDFS 中 HBase 存储状态

2. Ambari 部署

使用 Ambari 部署 HBase 可以说是一键操作，难点几乎都在 Ambari 工具本身部署上，以下步骤从无到有，简单介绍了 Ambari 自身部署和使用 Ambari 部署 HBase 的大概步骤：

- Step1 制定部署规划。
- Step2 准备硬件机器和 OS 环境。
- Step3 配置单机 OS 环境和集群环境。
- Step4 部署 Ambari-server。
- Step5 使用 Ambari-server 部署 HDFS、ZooKeeper、HBase。

例 1 请使用 Ambari 为 littleCstor 部署 HBase。

解 由于大数据平台涉及太多组件，故部署之前最好制定一个完备的部署计划，否则极有可能导致由于各机角色分配混乱而部署失败。

本题以第 3 章为前提，只给出 HBase 部署规划表（表 8-3）和部署效果图（图 8-12），具体部署过程请参见第 3 章。读者须注意，图 8-12 中 iclient0 到 ZooKeeper 连接（图中 Ø 表示）实际上并不存在，也就是 iclient0 并不需要向任何机器汇报心跳包，只有当 iclient0 需要使用 HBase 时，它才会主动连接 ZooKeeper，接着再由 ZooKeeper 告知客户端 HMaster 的详细地址。

表 8-3 littleCstor 上 HBase 部署规划

机 器	角 色	部署服务
cmaster2	主节点	HMaster
cslave0	从节点	HRegionServer
cslave1		
cslave2		
cslave3		
iclient0	客户端	HBase 客户端

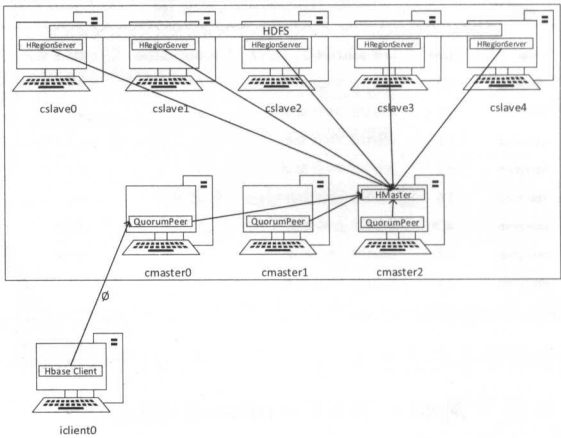


图 8-12 littleCstor 中的 HBase 拓扑图

8.2 HBase 接口

HBase 提供了诸多访问接口，下面简单罗列各种访问接口。

①Native Java API：最常规和高效的访问方式，适合 Hadoop MapReduce Job 并行批

处理 HBase 表数据。

②HBase Shell: HBase 的命令行工具, 最简单的接口, 适合管理、测试时使用。

③Thrift Gateway: 利用 Thrift 序列化技术, 支持 C++, PHP, Python 等多种语言, 适合其他异构系统在线访问 HBase 表数据。

④REST Gateway: 支持 REST 风格的 HTTP API 访问 HBase, 解除了语言限制。

⑤Pig: 可以使用 Pig Latin 流式编程语言操作 HBase 中的数据, 和 Hive 类似, 本质上最终也是编译成 MR Job 来处理 HBase 表数据, 适合做数据统计。

⑥Hive: 同 Pig 类似, 用户可以使用类 SQL 的 HiveQL 语言处理 HBase 表中数据, 当然最终本质依旧是 HDFS 与 MR 操作。

8.3 实战 HBase Shell

1. status

启动 HBase 后, 通过 shell 命令连接到 HBase, 并使用 status 命令查看 HBase 的运行状态, 确保 HBase 正常运行, 如下所示:

```
[allen@iclient0 ~]$ cd hbase-1.1.2/
[allen@iclient0 hbase-1.1.2]$ bin/hbase shell
hbase(main):001:0> status
4 servers, 0 dead, 0.7500 average load
hbase(main):002:0>
```

2. help

输入 help 命令可以查看有哪些 shell 命令以及参数选项, 从图 8-13 看出表名、行和列名等都要用单引号扩起来, 并以逗号隔开。

3. create

创建一个只包含一个列族 fam1 的表 tab1, 并通过 list 指令查看表是否创建成功, 如图 8-14 所示。

```
hbase(main):003:0> create 'tab1','fam1'
0 row(s) in 1.7720 seconds
=> Hbase::Table - tab1
hbase(main):004:0>
```

4. put

使用 `put` 命令向表中插入数据, 参数分别为表名、行名、列名和值, 其中列名前需要列族作为前缀, 时间戳由系统自动生成, 插入成功后通过 `scan` 命令查看表中的信息 (图 8-14)。

```
[allen@iclient0 hbase-1.1.2]$ bin/hbase shell
hbase(main):001:0> help
HBase Shell, version 1.1.2, rcc2b70cf03e3378800661ec5cab11eb43f4fe0fc, Wed Aug 26 20:11:27 PDT 2015
Type 'help "COMMAND"', (e.g. 'help "get"' the quotes are necessary) for help on a specific command.
Commands are grouped. Type 'help "COMMAND_GROUP"', (e.g. 'help "general"' for help on a command group.
```

进入HBase命令行接口

```
COMMAND GROUPS:
  Group name: general
  Commands: status, table_help, version, whoami

  Group name: ddl
  Commands: alter, alter_async, alter_status, create, describe, disable, disable_all, drop, drop_all, enable, enable_all, exists, get_table, is_disabled, is_enabled, list, show_filters

  Group name: namespace
  Commands: alter_namespace, create_namespace, describe_namespace, drop_namespace, list_namespace, list_namespace_tables

  Group name: dml
  Commands: append, count, delete, deleteall, get, get_counter, get_splits, incr, put, scan, truncate, truncate_reserve

  Group name: tools
  Commands: assign, balance_switch, balancer, balancer_enabled, catalogjanitor_enabled, catalogjanitor_run, catalogjanitor_switch, close_region, compact, compact_rs, flush, major_compact, merge_region, move, split, trace, unassign, wal_roll, zk_dump

  Group name: replication
  Commands: add_peer, append_peer_tableCFs, disable_peer, disable_table_replication, enable_peer, enable_table_replication, list_peers, list_replicated_tables, remove_peer, remove_peer_tableCFs, set_peer_tableCFs, show_peer_tableCFs

  Group name: snapshots
  Commands: clone_snapshot, delete_all_snapshot, delete_snapshot, list_snapshots, restore_snapshot, snapshot

  Group name: configuration
  Commands: update_all_config, update_config

  Group name: quotas
  Commands: list_quotas, set_quota

  Group name: security
  Commands: grant, revoke, user_permission

  Group name: visibility_labels
  Commands: add_labels, clear_auths, get_auths, list_labels, set_auths, set_visibility
```

普通操作

help选项, 查看所有支持命令

ddl类操作

常用工具集

配额类操作

实例

```
hbase> get 't1', "key\x03\x3f\xcd"
hbase> get 't1', "key\x03\x023\x011"
hbase> put 't1', "test\xef\xff", 'f1:', "\x01\x33\x40"
```

退出HBase

```
The HBase shell is the (J)Ruby IRB with the above HBase-specific commands added.
For more on the HBase Shell, see http://hbase.apache.org/book.html
hbase(main):002:0> quit
[allen@iclient0 hbase-1.1.2]$
```

图 8-13 help 命令结果的部分截图

```

hbase(main):001:0> create 'tab1','fam1'
0 row(s) in 3.3530 seconds

=> Hbase::Table - tab1
hbase(main):002:0> put 'tab1','row1','fam1:col1','val1'
0 row(s) in 0.3670 seconds

hbase(main):003:0> put 'tab1','row2','fam1:col2','val2'
0 row(s) in 0.0280 seconds

hbase(main):004:0> put 'tab1','row2','fam1:col3','val3'
0 row(s) in 0.0400 seconds

hbase(main):005:0> scan 'tab1'
ROW          COLUMN+CELL
 row1        column=fam1:col1, timestamp=1460924197509, value=val1
 row2        column=fam1:col2, timestamp=1460924205086, value=val2
 row2        column=fam1:col3, timestamp=1460924212501, value=val3
2 row(s) in 0.2000 seconds

hbase(main):006:0> get 'tab1','row2'
COLUMN      CELL
 fam1:col2   timestamp=1460924205086, value=val2
 fam1:col3   timestamp=1460924212501, value=val3
2 row(s) in 0.0480 seconds

hbase(main):007:0> delete 'tab1','row2','fam1:col2'
0 row(s) in 0.0570 seconds

hbase(main):008:0> scan 'tab1'
ROW          COLUMN+CELL
 row1        column=fam1:col1, timestamp=1460924197509, value=val1
 row2        column=fam1:col3, timestamp=1460924212501, value=val3
2 row(s) in 0.0310 seconds

hbase(main):009:0>

```

图 8-14 常用表操作

5. get

get 操作用来获取表中的一行数据，通过 delete 删除一行的数据。

6. 通过 disable 和 drop 命令删除 tab1 表

如图 8-15 所示。

```

hbase(main):009:0> disable 'tab1'
0 row(s) in 2.3470 seconds

hbase(main):010:0> drop 'tab1'
0 row(s) in 1.3060 seconds

hbase(main):011:0>

```

图 8-15 删除表操作

8.4 实战 HBase API

通过 Eclipse 创建一个新工程，并新建一个类 HBasicOperation，写代码前还要引入 Hadoop 开发所需要的 jar 包以及 HBase 包，完成上述操作后即可开发 HBase 应用。

①使用 `HBaseConfiguration.create()` 初始化 HBase 的配置文件, 并指定 HBase 使用的 ZooKeeper 的地址。然后实例化 `HBaseAdmin`, 该类用于对表的元数据进行操作并提供了基本的管理操作, 如下所示:

```
Configuration conf = HBaseConfiguration.create();
conf.set("HBase.zookeeper.quorum", "UbuntuSlave1, UbuntuSlave2, UbuntuSlave3");
HBaseAdmin admin = new HBaseAdmin(conf);
```

②`HBaseAdmin.createTable()` 可以用于创建一张新表, 该方法的参数为 `HTableDescriptor` 类, 用于描述表名和相关的列族。该方法的返回值为 `HTable` 类, 用于对表进行相关操作, 如下所示:

```
HTableDescriptor tableDescriptor = new HTableDescriptor("tab1".getBytes());
tableDescriptor.addFamily(new HColumnDescriptor("fam1"));
admin.createTable(tableDescriptor);
HTable table = new HTable(conf, "tab1");
```

③使用 `HTable.put()` 可以向表中插入数据, 该方法的参数为 `Put` 类, 该类初始化时可以传递一个行键, 表示向哪一行插入数据, 并通过 `Put.add()` 添加需要插入表中的数据, 如下所示:

```
Put putRow1 = new Put("row1".getBytes());
putRow1.add("fam1".getBytes(), "col1".getBytes(), "val1".getBytes());
table.put(putRow1);

System.out.println("add row2");
Put putRow2 = new Put("row2".getBytes());
putRow2.add("fam1".getBytes(), "col2".getBytes(), "val2".getBytes());
putRow2.add("fam1".getBytes(), "col3".getBytes(), "val3".getBytes());
table.put(putRow2);
```

④使用 `HTable.getScanner()` 可以获得某一个列族的所有数据 (其他重载详见 HBase 的 API 文档), 该方法返回 `Result` 类, `Result.getFamilyMap()` 可以获得以列名为 key, 值为 value 的映射表, 然后就可以依次读取相关的内容了。

```
for (Result row : table.getScanner("fam1".getBytes())) {
    System.out.format("ROW\t%s\n", new String(row.getRow()));
    for (Map.Entry<byte[], byte[]> entry : row.getFamilyMap("fam1".getBytes()).entrySet()) {
        String column = new String(entry.getKey());
        String value = new String(entry.getValue());
        System.out.format("COLUMN\tfam1:%s\t%s\n", column, value);
    }
}
```

⑤使用 `HBaseAdmin.disableTable()` 和 `HBaseAdmin.deleteTable()` 可以删除一张表。

```
admin.disableTable("tab1");
admin.deleteTable("tab1");
```

8.5 实战 HBase 之综例

假定有如下场景：①假定 MySQL 里有 member 表（表 8-4），要求使用 HBase 的 Shell 接口，在 HBase 中新建并存储此表。②简述 HBase 是否适合存储问题①中的结构化数据，并简单叙述 HBase 与关系型数据库的区别。

表 8-4 结构化表 member

身份 ID	姓名	性别	年龄	教育	职业	收入
201401	aa	0	21	e0	p3	m
201402	bb	1	22	e1	p2	l
201403	cc	1	23	e2	p1	m

HBase 是按列存储的分布式数据库，它有一个列族的概念，对应表 8-4，这里的列族应当是什么呢？这需要我们做进一步抽象，下面将姓名、性别、年龄这三个字段抽象为个人属性（personalAttr），教育、职业、收入抽象为社会属性（socialAttr），personalAttr 列族包含 name、gender 和 age 三个限定符；同理 socialAttr 下包含 edu、prof、inco 三个限定符，表 8-5 是针对表 8-4 的进一步逻辑抽象。

表 8-5 HBase 里 member 表的逻辑模型

Key 行键	Value 列键					
	列族 personalAttr			列族 socialAttr		
身份 ID	姓名	性别	年龄	教育	职业	收入
201401	aa	0	21	e0	p3	M
201402	bb	1	22	e1	p2	L
201403	cc	1	23	e2	P1	M

按上述思路，iclient0 上依次执行如下命令：

[allen@iclient0 hbase-1.1.2]\$ bin/hbase shell#进入 HBase 命令行
HBase(main):001:0> list#查看所有表
HBase(main):002:0> create 'member','id','personalAttr','socialAttr'#创建 member 表
HBase(main):003:0> list
HBase(main):004:0> scan 'member'#查看 member 内容
HBase(main):005:0> put 'member','201401','personalAttr:name','aa'#向 member 表中插入
数据
HBase(main):006:0> put 'member','201401','personalAttr:gender','0'
HBase(main):007:0> put 'member','201401','personalAttr:age','21'
HBase(main):008:0> put 'member','201401','socialAttr:edu','e0'


```

HBase(main):009:0> put 'member','201401','socialAttr:job','p3'
HBase(main):010:0> put 'member','201401','socialAttr:income','m'
HBase(main):011:0> scan 'member'
HBase(main):012:0> disable 'member'                                #废弃 member 表
HBase(main):013:0> drop 'member'                                   #删除 member 表
HBase(main):014:0> quit

```

其实 HBase 里的数据依旧可以看成<Key,Value>对,只是它的 Value 可以是一个 List,即<Key,ValueList>, <Key,Value1,...,ValueN>,如表中的<ID,[personalAttr,socialAttr]>,而每个列族也是一个 List,比如列族 personalAttr 包含三个限定符 name, gender, age。读者也可以只定义一个列族,比如列族 info,此列族下包含六个限定符。

显然表 8-4 键 ID 数量众多,且其结构定义完整,事实上 HBase 并不适合存储这类结构化数据,HBase 设计之初是为了存储互联网上大量的半结构化数据,比如本题中用户甚至都可以 put 'member','201401','socialAttr:country','china',而表中并没有定义 country 字段,但 HBase 里可以随意插入,这是它的巨大优势,这是问题②前一问答案,针对后一问,下面简单罗列 HBase 和关系型数据库的区别。

HBase 只提供字符串这一种数据类型,其他数据类型的操作只能靠用户自行处理,而关系型数据库有丰富的数据类型;HBase 数据操作只有很简单的插入、查询、删除、修改、清空等操作,不能实现表与表关联操作,而关系型数据库有大量此类 SQL 语句和函数;HBase 基于列式存储,每个列族都由几个文件保存,不同列族的文件是分离的,关系型数据库基于表格设计和行模式保存;HBase 修改和删除数据实现上是插入带有特殊标记的新记录,而关系型数据库是数据内容的替换和修改;HBase 为分布式而设计,可通过增加机器实现性能和数据增长,而关系型数据库很难做到这一点。

8.6 实战 HBase 之使用 MapReduce 构建索引

8.6.1 索引表蓝图

HBase 索引主要用于提高 HBase 中表数据的访问速度,有效地避免了全表扫描(多数查询可以仅扫描少量索引页及数据页,而不是遍历所有的数据页)。HBase 中的表根据行键被分成了多个 Regions,通常一个 Region 的一行都会包含较多的数据,如果以列值作为查询条件,就只能从第一行数据开始往下查找,直到找到相关数据为止,这显然很

低效。相反，如果将经常被查询的列作为行键、行键作为列重新构造一张表，即可实现根据列值快速地定位相关数据所在的行，这就是索引。显然索引表仅需要包含一个列，所以索引表的大小和原表比起来要小得多，如图 8-16 给出了索引表与原表之间的关系。从图 8-16 可以看出，由于索引表的单条记录所占空间比原表要小，所以索引表的一个 Region 与原表相比，能包含更多条记录。

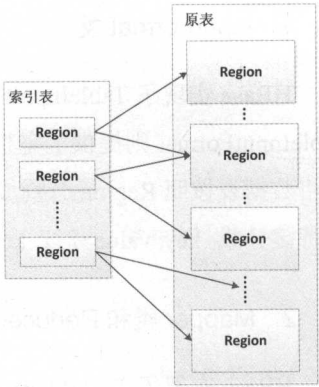


图 8-16 索引表与原表关系

假设 HBase 中存在一张表 heroes，内容见表 8-6。则根据列 info:name 构建的索引表如图 8-17 所示。

表 8-6 heroes 表的逻辑视图

行键	列族 info		
	name	email	power
1	peter	peter@heroes.com	absorb abilities
2	hiro	hiro@heroes.com	bend time and space
3	sylar	sylar@heroes.com	know how things work
4	claire	claire@heroes.com	heal
5	noah	noah@heroes.com	cath the people with abilities

HBase 会自动将生成的索引表加入表 8-6 所示的结构中，从而提高搜索的效率。

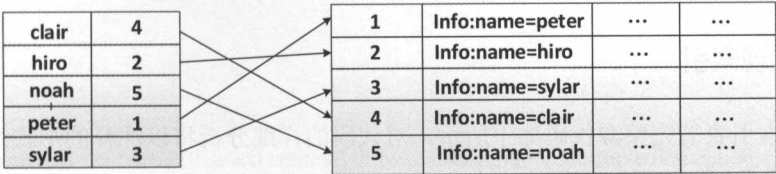


图 8-17 heroes 索引表示意

8.6.2 HBase 和 MapReduce

HBase 中的表通常是非常大的，并且还可以不断增大，所以为表建立索引的工作量也是相当大的。为了解决类似的问题，HBase 集成了 MapReduce 框架，用于对表中的大量数据进行并行处理。前文已经对 MapReduce 的处理过程进行了详细的介绍，根据这些处理过程，HBase 为每个阶段提供了相应的类来处理表数据。

1. InputFormat 类

HBase 实现了 `TableInputFormatBase` 类, 该类提供了对表数据的大部分操作, 其子类 `TableInputFormat` 则提供了完整的实现, 用于处理表数据并生成键值对。 `TableInputFormat` 类将数据表按照 `Region` 分割成 `split`, 即有多少个 `Regions` 就有多个 `splits`。然后将 `Region` 按行键分成 `<key, value>` 对, `key` 值对应于行键, `value` 值为该行所包含的数据。

2. Mapper 类和 Reducer 类

HBase 实现了 `TableMapper` 类和 `TableReducer` 类, 其中 `TableMapper` 类并没有实现具体的功能, 只是将输入的 `<key, value>` 对的类型分别限定为 `ImmutableBytesWritable` 和 `Result`。 `IdentityTableMapper` 类和 `IdentityTableReducer` 类则是上述两个类的具体实现, 其和 `Mapper` 类及 `Reducer` 类一样, 只是简单地将输入 `<key, value>` 对输出到下一个阶段。

3. OutputFormat 类

HBase 实现的 `TableOutputFormat` 将输出的 `<key, value>` 对写到指定的 HBase 表中, 该类不会对 WAL (Write-Ahead Log) 进行操作, 即如果服务器发生故障将面临丢失数据的风险。可以使用 `MultipleTableOutputFormat` 类解决这个问题, 该类可以对是否写入 WAL 进行设置。

8.6.3 实现索引

构建索引表的完整源代码如下所示, 对代码的详细分析将以注释的形式给出。运行程序需要设置运行参数, 分别为表名、列族和需要建索引的列 (列必须属于前面给出的列族)。如要对 `heroes` 中的 `name` 和 `email` 列构建索引, 则运行参数应设为: `heroes info name email`。

```
import java.io.IOException;
import java.util.HashMap;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.HBase.HBaseConfiguration;
import org.apache.hadoop.HBase.client.Put;
import org.apache.hadoop.HBase.client.Result;
import org.apache.hadoop.HBase.client.Scan;
import org.apache.hadoop.HBase.io.ImmutableBytesWritable;
import org.apache.hadoop.HBase.util.Bytes;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.mapreduce.Job;
```

```

import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.util.GenericOptionsParser;
public class IndexBuilder {
    //索引表唯一的一列为 INDEX:ROW, 其中 INDEX 为列族
    public static final byte[] INDEX_COLUMN = Bytes.toBytes("INDEX");
    public static final byte[] INDEX_QUALIFIER = Bytes.toBytes("ROW");
public static class Map extends Mapper<ImmutableBytesWritable, Result, ImmutableBytesWritable,
Writable> {
    private byte[] family;
    //存储了“列名”到“表名-列名”的映射
    //前者用于获取某列的值, 并作为索引表的键值; 后者用于作为索引表的表名
    private HashMap<byte[], ImmutableBytesWritable> indexes;
    protected void map(ImmutableBytesWritable rowKey, Result result, Context context) throws
IOException, InterruptedException {
        for(java.util.Map.Entry<byte[], ImmutableBytesWritable> index : indexes.entrySet()) {
            byte[] qualifier = index.getKey();    //获得列名
            ImmutableBytesWritable tableName = index.getValue();    //索引表的表名
            byte[] value = result.getValue(family, qualifier);    //根据“列族:列名”获得元素值
            if (value != null) {
                //以列值作为行键, 在列“INDEX:ROW”中插入行键
                Put put = new Put(value);
                put.add(INDEX_COLUMN, INDEX_QUALIFIER, rowKey.get());
                //在 tableName 表上执行 put 操作
                //使用 MultiOutputFormat 时, 第二个参数必须是 Put 或 Delete 类型
                context.write(tableName, put);
            }
        }
    }
}

//setup 为 Mapper 中的方法, 该方法只在任务初始化时执行一次
protected void setup(Context context) throws IOException, InterruptedException {
    Configuration configuration = context.getConfiguration();
    //通过 configuration.set()方法传递参数, 详见下面的 configureJob 方法
    String tableName = configuration.get("index.tablename");
    String[] fields = configuration.getStrings("index.fields");
    //fields 内为需要做索引的列名
    String familyName = configuration.get("index.familyname");
    family = Bytes.toBytes(familyName);
    //初始化 indexes 方法
    indexes = new HashMap<byte[], ImmutableBytesWritable>();
    for(String field : fields) {
        // 如果给 name 做索引, 则索引表的名称为“heroes-name”
        indexes.put(Bytes.toBytes(field),new ImmutableBytesWritable(Bytes.toBytes(tableName + "-" +
field)));
    }
}

```

```

    }
}

public static Job configureJob(Configuration conf, String [] args) throws IOException {
    String tableName = args[0];
    String columnFamily = args[1];
    System.out.println("*****" + tableName);
    //通过 Configuration.set()方法传递参数
    conf.set(TableInputFormat.SCAN, TableMapReduceUtil.convertScanToString(new Scan()));
    conf.set(TableInputFormat.INPUT_TABLE, tableName);
    conf.set("index.tablename", tableName);
    conf.set("index.familyname", columnFamily);
    String[] fields = new String[args.length - 2];
    for(int i = 0; i < fields.length; i++) {
        fields[i] = args[i + 2];
    }
    conf.setStrings("index.fields", fields);
    conf.set("index.familyname", "attributes");
    //配置任务的运行参数
    Job job = new Job(conf, tableName);
    job.setJarByClass(IndexBuilder.class);
    job.setMapperClass(Map.class);
    job.setNumReduceTasks(0);
    job.setInputFormatClass(TableInputFormat.class);
    job.setOutputFormatClass(MultiTableOutputFormat.class);
    return job;
}

public static void main(String[] args) throws Exception {
    Configuration conf = HBaseConfiguration.create();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if(otherArgs.length < 3) {
        System.err.println("Only " + otherArgs.length + " arguments supplied, required: 3");
        System.err.println("Usage:  IndexBuilder  <NAME>  " + "<FAMILY>  <ATTR> [<ATTR> ...]");
        System.exit(-1);
    }
    Job job = configureJob(conf, otherArgs);
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}

```

习 题

1. 既然已经有了 HDFS 来持久化“大数据”，为什么还要 HBase？
2. 我们知道，数据库的核心技术之一就是索引，试问在 HBase 中，是动态建索还是静态建索？是一级索引还是多级索引？
3. 简述 HBase 功能作用及其体系架构。
4. 在 HBase 中，“.META.”是什么？存储在哪里？表格呢？
5. 在 HBase 中，为什么需要 ZooKeeper？当 HMaster 宕机后，HBase 是否可继续提供读服务？
6. 简述手工部署 HBase、使用 Ambari 部署 HBase 的步骤。
7. 简述 HBase 访问接口。
8. 简述使用 Maven 和不使用 Maven 时，HBase 开发环境搭建部署。
9. 简述常见的 HBase 调优技术。
10. 在大型系统中，HBase 最常使用的场景是什么？
11. 在大型系统中，如何使用 HBase 提供在线实时服务？
12. 在大型集群中，如何确保 HBase 集群本身安全性？如何确保 HBase 数据安全性？
13. 简述 HRegionServer 分裂过程。
14. 简述 HBase 和 Hive 区别。

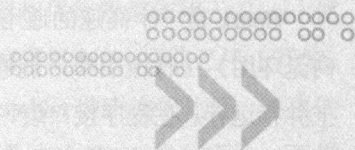
参考文献

- [1] <http://hbase.apache.org/>
- [2] <http://hbase.apache.org/book.html>

第9章

内存型计算框架 Spark

HADOOP
BEING DIGITAL



Spark Streaming
实时流式计算

Spark SQL
结构化查询语言

Spark MLlib
机器学习库

Spark GraphX
图计算库

Spark Resilient Distributed Dataset (RDD)
弹性分布式数据集

Spark 是一个基于内存的计算框架，它可以在 Hadoop 集群上运行。Spark 的主要组件包括：Spark Core、Spark SQL、Spark Streaming、Spark MLlib 和 Spark GraphX。Spark Core 是 Spark 的核心，它负责管理集群资源、调度任务、以及处理数据。Spark SQL 是 Spark 的查询引擎，它允许用户使用 SQL 语言来查询存储在 Hadoop 中的数据。Spark Streaming 是 Spark 的流处理引擎，它允许用户实时处理来自 Hadoop 的数据流。Spark MLlib 是 Spark 的机器学习库，它提供了一系列机器学习算法，可以用于对存储在 Hadoop 中的数据进行分析。Spark GraphX 是 Spark 的图计算库，它提供了一系列图计算算法，可以用于对存储在 Hadoop 中的图数据进行分析和处理。

Spark 是一个由加州伯克利分校开发的内存型计算框架^[1]，设计之初是为了处理迭代型机器学习任务，目前 Spark 上已经集成了数据仓库、流处理、图计算等多种实用工具，是大数据领域完整的全栈计算平台。本章第一节重点讲述 Spark 基本理论，接着在实战环节给出 Spark 核心弹性分布式数据集（Resilient Distributed Datasets，RDD）大量编程实例，以让读者以最快方式学习 Spark。

9.1 Spark 简介

作为目前使用最广的内存型计算框架，Spark^[2]在设计上有其独到之处。本节即从理论上讲述 Spark 框架、运行模式和工作机制。在讲述过程中，编者会将 Spark 和 Tez、MapReduce 进行对比，以说明 Spark 为何如此优秀。

9.1.1 基础概念

Spark 于 2009 年诞生于伯克利大学 AMPLab，最初它只是伯克利大学的研究性项目。2010 年时 Spark 正式开源，2013 年，伯克利将其捐赠给 Apache 基金会。仅仅一年之后的 2014 年，Spark 已成为 Apache 基金的顶级项目，目前 Spark 已广泛应用于工业界。Spark 之所以普及得如此之快，和当前大数据环境以及 Spark 优秀的性能是分不开的。

1. Spark 及其生态圈

Spark 是一个高速的通用型集群计算框架，其内部内嵌了一个用于执行 DAG 的（有向无环图）工作流引擎，能够将 DAG 类型的 Spark-App 拆分成 Task 序列并在底层框架上并行运行。在程序接口层，Spark 为当前主流语言都提供了编程接口，如用户可以使用 Java、Scala、Python、R 等高级语言直接编写 Spark-App。此外，在核心层之上，Spark 还提供了诸如 SQL、Mllib、GraphX、Streaming 等专用组件（图 9-1），这些组件内置了大量专用算法，充分利用这些组件，能够大大加快 Spark-App 开发进度。

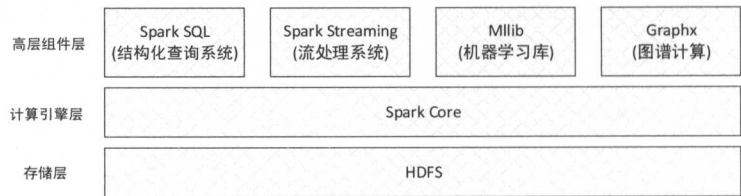


图 9-1 Spark 生态圈

一般称 Spark Core 为 Spark，从图 9-1 可以看出，Spark（即 Spark Core）处于存储层和高层组件层的中间，定位为计算引擎，核心功能是并行化执行用户提交的 DAG 型 Spark-App。正如图中所示，目前 Spark 生态圈主要包括 Spark Core 和基于 Spark Core 的独立组件（SQL、Streaming、Mllib 和 Graphx）。

1) HDFS

从 Hadoop 诞生至今，计算框架从无发展到 Mesos，再到 YARN，并行化范式由 M-S-R 发展到 DAG 型 M-S-R，唯一未曾改变^[1]的就是 HDFS，虽然 HDFS 在小文件存储上的确存在性能缺陷，但无疑 HDFS 已经成为分布式存储事实标准，故在 Spark 开发过程中，伯克利并未开发一套独立分布式底层存储系统^[2]，而是直接使用了 HDFS。当然，Spark 也可以运行在本地文件系统或内存型文件系统（如 Spark 原生支持 Tachyon）上，不过在典型应用模式中，持久层依旧是 HDFS。

2) Spark

Spark（即图中的 Spark Core）的核心功能是将用户提交的 DAG 型的 Spark-App 任务拆分成 Task 序列并在底层框架上并行执行。如用户提交的某作业经抽象后执行操作流为“Map→Reduce→Reduce→Map→Reduce”，使用 Spark 时只需几条简单语句即可完成处理。为实现此功能，Spark 提供了任务调度、任务分解、内存管理、故障恢复、I/O 等功能。编程接口方面，Spark 通过 RDD 将框架功能和操作函数优雅地结合起来，大大方便了用户编程，读者将会在 9.4 节看到，Spark 编程如此简单。

3) Spark SQL

在 Spark 早期开发过程中，为支持结构化查询，开发人员在 Spark 和 Hive 基础上开发了结构化查询模块，称为 Shark。不过由于 Shark 的编译器和优化器都依赖于 Hive，使得 Shark 不得不维护一套 Hive 分支，执行速度也受到 Hive 编译器制约，目前已停止开发。

Spark SQL 的功能和 Shark 类似，不过它直接使用了 Catalyst 做查询优化器，不再依赖 Hive 解析器，其底层也直接使用 Spark 作为执行引擎。通过对 Shark 进行重构，不仅使得用户能够直接在 Spark 上书写标准 SQL 语句，大大加快 SQL 执行速度，还为 Spark SQL 发展开拓了广阔空间。

4) Spark Streaming

Streaming 是一个基于 Spark 内核开发的一套可扩展、高通用、容错的实时数据流处理框架。Streaming 的数据源可以是 Kafka、Flume、Twitter、ZeroMQ、Kinesis 甚至是 TCP Socket。在 Streaming 中，可以通过复杂的高层函数（如 map、reduce、join、window 等的组合）直接处理这些数据。在完成数据转换后（过滤、整合等），Streaming 会将数据自动输出到持久层，目前 Streaming 支持的持久层为 HDFS、数据库和实时控制台。特别

地, 用户可以在数据流上使用 `mllib` 和 `graphx` 里的所有算法。

5) MLlib

MLlib 是 Spark 上的一个机器学习库, 其设计目标是开发一套高可用、高扩展的并行机器学习库, 以方便用户直接调用。目前 MLlib 下已经开发了大量常见机器学习算法 (如 `classification`、`regression`、`clustering`、`collaborative filtering`、`dimensionality reduction` 等), 为方便用户使用, MLlib 还提供了一套实用工具集 (如低层的优化原语工具类、高层管道工具类、矩阵转换工具类等)。

用户可直接调用 MLlib 库里已经开发好的机器学习算法, 比如编者直接使用 MLlib 里的 SVM 来并行化训练 Parkinson 数据集, 事实证明用 Spark 训练时性能非常优秀, 在 9.6 节, 编者将详细分析 MLlib 里的 SVM。

6) Graphx

Graphx 是 Spark 上一个图处理和图并行化计算的全新组件。为实现图计算, Graphx 引入了一个继承自 RDD 的新抽象数据集——Graph, 该类是一个有向的带权图谱, 用户可以自定义 Graph 的顶点和边属性。目前, Graphx 已经开发了一系图基本操作 (如 `subgraph`、`joinVertices`、`aggregateMessages` 等) 和一些优化的 Pregel 变体 API。用户只要将样本数据填充到 I/O 类 (Graphx 提供, 如矩阵), 然后直接调用图算法, 即可完成图的并行化计算。

2. Spark 计算模型初探

通过上面的介绍, 读者应该能够大致了解 Spark 的主要功能及其所处位置, 不过好像 MapReduce 框架也基于 HDFS 的 M-S-R 型计算框架, 那到底 Spark 和 Hadoop、Spark 和 MapReduce 框架有何关系? 下面即讲述两者区别。

1) 并行化范式最小单元

尽管实用型的数据处理流表现出的处理过程具有一定的复杂性, 不过, 仔细分析其中并行化部分, 会发现基本的并行化方式相当固定, 即 M 范式、M-S-R 范式、BSP 范式 (图 9-2)。实际上, 所有的并行化任务^[1]都是这三大范式或这三大范式的组合范式, Spark 和 MapReduce 框架一样, 都是 M-S-R 范式的代码实现。

2) Hadoop 和 MapReduce

Hadoop 是大数据的一整套解决方案体, 它主要包括分布式存储 HDFS、分布式资源管理器 YARN 和分布式计算引擎 MapReduce 框架。就像一个学校可以包含三个学院一样, Hadoop 只是这三个组件的合称。

实际上, Hadoop 里 MapReduce 框架就是 Apache 针对 M-S-R 范式的代码实现, 显然, 并不是只允许 Apache 一家能对 M-S-R 范式进行代码实现, 其他组织也可以。伯克利在充分借鉴其他框架优点、避开其缺点的情况下, 开发出了 Spark 框架。

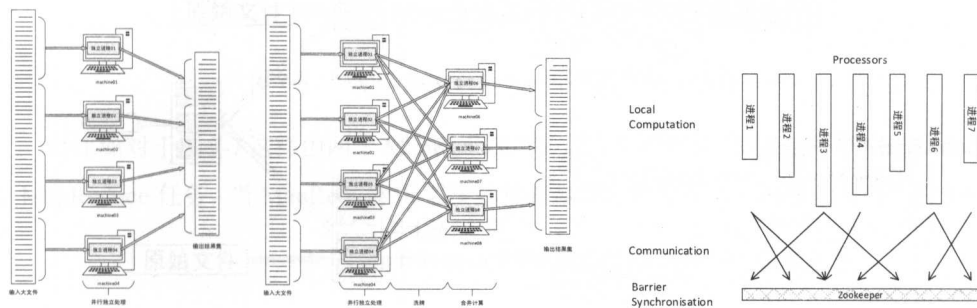


图 9-2 三大并行范式示意图

3) Spark 计算模型

当 MapReduce 框架执行类似 “Map→Reduce→Map→Reduce” 型任务时, 其实质上还是将任务拆分成两个 MapReduce, 集群中某特定处理节点上会依次启动四个进程处理该任务。Spark 原生支持 DAG 型作业, 并且集群中某特定处理节点上, 整个过程始终只有一个 Executor 进程, 其内部使用线程来处理各模块。

比如现有这样一场景: 对三个原始文件 rawFilez、rawFiley、rawFilex, 按要求将 rawFiley 自身一次处理后和 rawFilex 进行 union。接着将 union 后的结果和 rawFilez 按 key 值洗牌后的结果进行 join, 记 join 后输出的结果文件为 resultFile。

对于上述问题, 实际上 MapReduce 和 Spark 处理机制是一样的, 都是该分几步就分几步, 不同的是, Spark 将这些步骤合并到了一个平台下, MapReduce 框架则需要使用 MR-App 串才可解决。

(1) 采用 MapReduce

当使用 MapReduce 处理上述问题时, 需要四个 MapReduce 串联起来才能完成任务, 第一个单 Map 将 rawFiley 转为 tmpResult1, 第二个单 Map 将 tmpResult1 和 rawFilex 转换成 tmpResult2, 第三个完整 MapReduce 将 rawFilez 转换成 tmpResult3。最后, 再写一个 MapReduce 将 tmpResult2 和 tmpResult3 结合并将结果转换成最终结果 resultFile, 各 MapReduce 串及其依赖关系如图 9-3 所示。

由于 Hadoop 不支持 DAG 型 MapReduce, 故当执行上述 MapReduce 串时, 需要依赖第三方软件或附加代码将这四个 MR-App 粘合成一个处理流, 常见的粘合工具为 Tez、Oozie、Java、Shell、Python, 推荐使用 Oozie 和 Tez。

(2) 采用 Spark

当使用 Spark 处理上述问题时, 只需四条语句即可完成处理, 和 Hadoop 不同, Spark

原生支持 DAG 型 M-S-R 任务,且其内置的 DAGSchedule 还能自动对上述任务进程优化,图 9-4 为使用 Spark 时程序执行步骤。

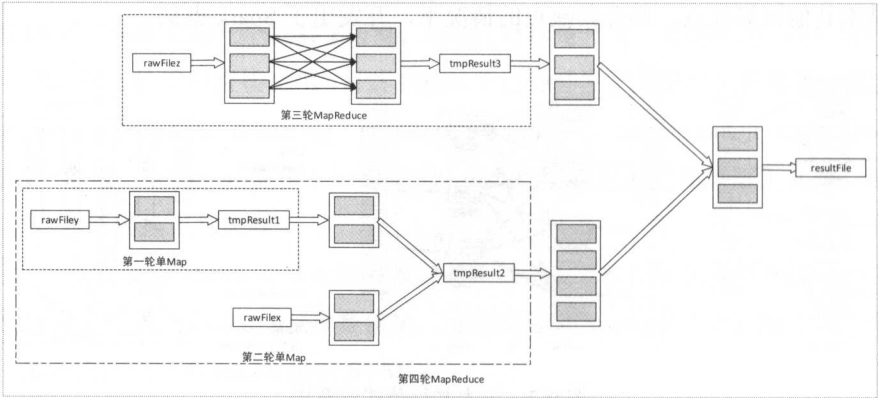


图 9-3 使用 MapReduce 串处理

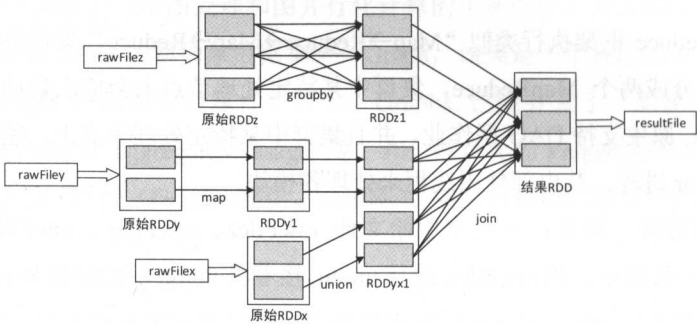


图 9-4 使用 Spark 处理

3. Spark 和 MapReduce 比较

Spark 框架和 MapReduce 框架都是并行化范式 M-S-R 的代码实现,两者各有其优缺点,下面是两个框架的比较。

1) M-S-R 执行次数

单个 MR-App 只执行一次 M-S-R,而单个 Spark-App 则可执行多次 M-S-R。实际上 Spark-App 都可拆分成一个个独立的 MR-App。简单地说,Spark 内嵌 DAG 执行器,可原生高效执行 DAG 型 MapReduce 任务。

2) 中间结果

当 MR-App 在 MapReduce 框架上执行时,框架会将 Map 的中间结果输出到磁盘。比如对于图 9-5 中的单 MapReduce 的任务,实际执行时,在 Map 和 Reduce 中间会有一次写磁盘操作(图 9-6)。

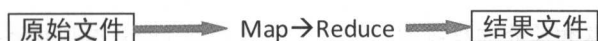


图 9-5 单 MapReduce



图 9-6 MapReduce 框架执行单 MR-App 示意图

再比如对于图 9-7 这样由两个 MapReduce 串起来(可通过 Shell 粘合两个 MapReduce)的 MapReduce 任务, 当 MapReduce 框架实际执行时, 其中会有三次写磁盘操作(图 9-8)。

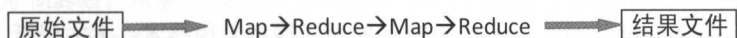


图 9-7 两个 MapReduce



图 9-8 MapReduce 框架依次执行两个 MR-App 示意图

和 MapReduce 框架相比, 当 Spark 执行图 9-5 中的单 MapReduce 任务时, 其 Map 操作结果直接存至内存, 不存入本地磁盘, 实际执行过程就是图 9-5; 又由于 Spark 内嵌了 DAG (有向无环图) 调度器, 当 Spark 执行图 9-7 中的 DAG 型 MapReduce 任务时, 其 Reduce 结果不存入 HDFS, 依旧存至内存, 实际执行过程就是图 9-7。

其实, MapReduce 框架上的 Tez 也提供了类似 DAGSchedule 功能, 只不过 Spark 将 DAGSchedule 嵌入内核, 使之执行更加高效。类似的 Tez 和 Spark 这种 DAG 执行器还有 Dryad。

3) 数据操作灵活性

Spark 将数据和并行化操作高度抽象到 RDD 中, 通过 RDD, 用户可以精确到每条记录, 可以控制不同节点上的分区数据, 具有非常大的灵活性。对于 MapReduce 框架, 由于没有一个类似 RDD 的抽象数据存储层, MapReduce 框架的数据操作不够灵活, 比如若用户需要单独处理第五个 Partition 上的第三行数据, 可能需要自己编写大量的代码, 而 Spark 则只需一行代码即可获取到该数据引用。

4) 排序策略

默认情况下, MapReduce 框架在 Map 之后、Reduce 之前、Reduce 之后, 其数据都是按 key 排序的。实际上, 这个过程相当耗时, 比如数据量稍大时, 就需要使用外排序; 和 MapReduce 框架默认使用排序不同, 除非显示声明开启排序, 否则 Spark 在这几个阶段都不会进行排序, 不过, 若用户需要某阶段的数据进行排序, 只需要调用 `RDD.sortByKey()` 即可, 操作非常简单。

5) 启动进程数和任务响应时间

Hadoop 上的 MapReduce 启动非常慢, 有时甚至启动时间比执行时间还长, 这是因为 Hadoop 的 MapReduce 框架设计之初就是为批处理而设计的, 其运行的任务可能长达数周或数月之久, 故较长启动时间完全可以接受。

Spark 则为交互式或实时并行化机器学习而设计, 其内部通过线程复用来避免进程或线程启动和切换开销, 因此, 和 MapReduce 框架相比 Spark 任务响应非常快。

假定有十台机器在同时计算上述数据 (图 9-3), cslave5 为其中一台, 当使用 MapReduce 框架处理时, cslave5 上会先后启动 5 个进程来计算任务; 当使用 Spark 处理时, 只有一个 Executor 进程。Executor 内部采用了线程复用技术, 大大降低了进程切换, 任务切换带来的时间开销。

其实 Spark 是 MapReduce 框架的替代方案, Spark 可直接读取 Hive, HDFS 上的数据, 可以狭义地认为其为解决 MapReduce 框架不支持 DAG 型 M-S-R 范式发展而来。图 9-9 可说明 Spark 和 MapReduce 之间关系, 从图中可以看出, Spark 框架是在 MapReduce 框架、Tez 引擎的基础上发展而来的。

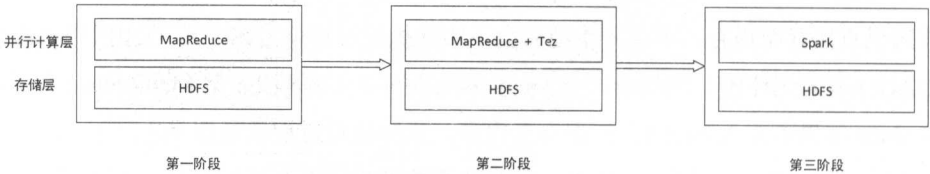


图 9-9 MapReduce 和 Spark 发展阶段

9.1.2 体系架构

同大多数分布式系统一样, Spark 也采用两层的 master/slave 架构, 第一层为集群资源管理层, 第二层为 Spark-App 执行层。

在集群环境下, 为管理好单机资源, 需要在集群中每台机器上部署一个独立进程, 一般称这类进程为从进程 (slave); 为整合整个集群内所有单机资源, 还需要一个管理进程来管理这些 slave, 一般称该角色为 master。当前 Spark 支持的集群资源管理器主要为 Standalone 模式和 ThirdPlatform 模式。

Standalone 模式指的是伯克利为 Spark 原生开发的 Master/Worker 资源管理器, ThirdPlatform 指的是当前主流的第三方集群资源管理器 YARN 和 Mesos。这两类管理主要区别是资源划分粒度粗细不同, Standalone 管理器资源划分颗粒更细, ThirdPlatform (YARN) 的优势在于很容易将不同组件集成到一个平台上。下面根据这两种不同的集群

资源管理器，分两部分讲述 Spark 体系架构。

1. Standalone 模式

Standalone 资源管理器是伯克利为 Spark 独立开发的一套资源管理器，主要特点是资源划分颗粒细，执行效率高，是目前对 Spark 支持度最好的资源管理器。

1) 集群资源管理层

典型的集群管理模式都采用 master/slave 架构，Standalone 资源管理器也不例外。其主服务就称为 Master，从服务则称为 Worker。单机上的驻守进程 Worker 主要负责管理本机资源。集群资源整合者 Master 进程则主要负责汇总各 Worker 进程汇报的单机资源（图 9-10）。

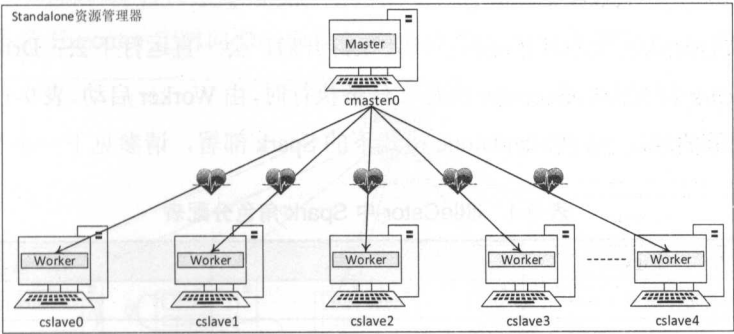


图 9-10 Standalone 集群资源管理器

实际上，Master 进程还提供 Web-UI 功能，Spark-App 注册功能，任务调度功能。特别是 Spark-App 注册功能，它是客户提交程序的入口。Worker 进程还提供启动和监管 Executor 进程功能。

2) 应用程序执行层

在集群上执行 Spark-App 时，其执行过程依旧采用 master/slave 架构，即 master 机负责控制程序整体执行流，slave 机负责具体执行某个任务（由 master 分配给自己）。

Spark-App 执行层就采用 master/slave 架构，其 master 服务称为 Driver，slave 服务称为 Executor。Driver 进程主要负责控制程序整个执行流（图 9-11），各个 Executor 进程负责并行执行某个具体任务（由 Driver 分配）。

实际上，在 iclient0 提交 Spark-App 之前，集群中并不存在 Driver 和 Executor 进程，只存在 Master/Worker 进程。当 client 向 Master 任务后，Worker 才会启动 Executor 来执行用户任务。

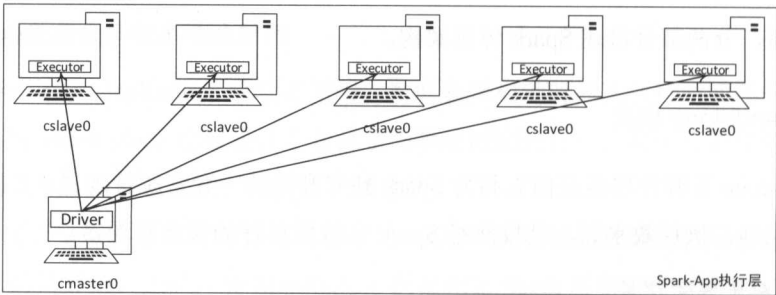


图 9-11 Spark-App 执行层

3) 集群部署

当采用 Standalone 模式时，所谓的部署 Spark，就是指部署 Standalone 资源管理器和 Client（Executor 由 Worker 启动，Driver 则是在 Client 里）。

Master 和 Worker 作为常驻进程，只要集群开启，会一直运行下去；Driver 进程则是在 Client 提交任务时出现，Executor 也是在任务执行时，由 Worker 启动。表 9-1 为 littleCstor 中 Spark 角色分配表，关于 Standalone 模式下的 Spark 部署，请参见下一小节。

表 9-1 littleCstor 中 Spark 角色分配表

角 色	主 机	进程名
资源管理主服务	cmaster0	Master
资源管理从服务	cslave0	Worker
	cslave1	
	cslave2	
	cslave3	
	cslave4	
客户端	iclient0	Driver（非驻守）
	iclient1	

4) 应用程序执行过程

当集群资源由 Standalone 管理时，若 Client 欲在集群上运行 Spark-App，则 Client 须首先向 Master 注册本应用程序；接着 Master 在确定该应用合法后，会通知各个 Worker 启动 Executor；在启动好 Executor 后，这些 Executor 会向 Driver（client）注册自己；最后 Driver 里的 TaskScheduler 会向这些 Executor 分发任务，而这些 Executor 则作为 Spark-App 实际执行者来执行 Driver 分配给自己的任务并将结果返回给 Driver。

请读者务必注意，每个 Spark-App 都包含一系列 Executor，每个 Executor 则只属于一个 Spark-App。图 9-12 为正在运行两个 Spark-App 的 Spark 集群概况图，从图中可以看出，集群资源管理者为 Master 和 Worker，两者之间为 master/slave 架构；Spark-App 执行

者为 Driver 和 Executor，两者之间也为 master/slave 架构。iclient0 提交的 Spark-App 拥有四个 Executor，iclient1 提交的 Spark-App 也是四个 Executor。隶属于不同 Spark-App 的 Executor 间毫无关联。

以 iclient0 提交的 Spark-App 为例，下面给出任务执行过程，为尽量保持图片简洁，编者只在 cslave0, cslave1 上标明了步骤③，cslave2、3、4 中并未标明此步，其标记与 cslave0、cslave1 相同。

- Step1 Client 向 Master 注册应用程序（图中步骤①）。
- Step2 Master 指示 Worker 启动 Executor（图中步骤②）
- Step3 Worker 启动从属于本应用的 Executor（图中步骤③）
- Step4 各个 Executor 向 Driver 的 ScheduleBackend 注册（图中步骤④）
- Step5 TaskSchedule 将任务分发到各 Executor 上执行（图中步骤④）
- Step6 各个 Executor 定时向 Driver 汇报本 Task 执行进度（图中步骤④）

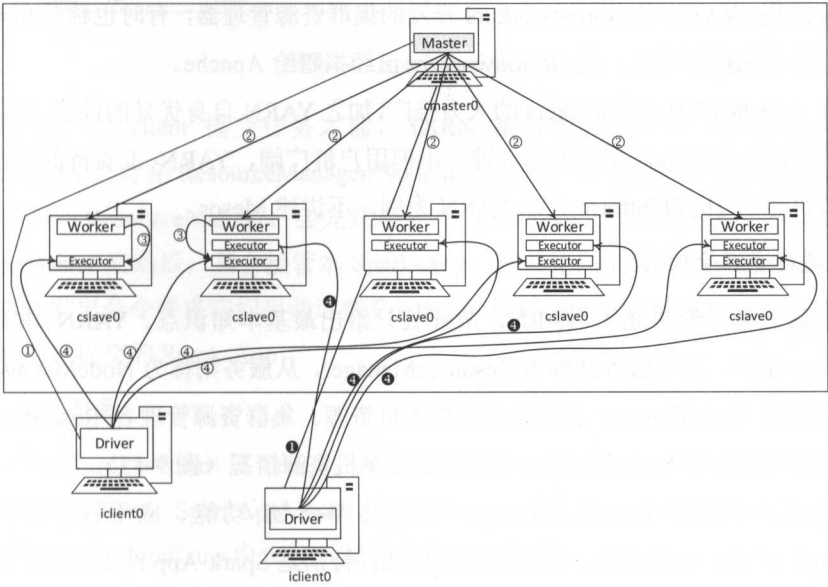


图 9-12 正在运行两个 Spark-App 的 Spark 集群

同理，iclient1 提交的 Spark-App 也主要包含这六步，不过由于标记太多，影响图清晰度，编者只标记了步骤①，步骤②和步骤④，步骤③并未画出，请读者参考步骤③。

综上所述，在 Standalone 模式中，当 Spark 集群中未运行任何 Spark-App 时，集群中仅存在 Master 和 Worker 进程。不过，当 client 向 Spark 集群（实际上是进程 Master）提交 Spark-App 时（如 PI），集群中会出现多个 Executor 和一个 Driver 进程。这两类进程（Master/Worker, Driver/Executor）都采用 master/slave，其中 Master/Worker 负责资源管理，

Driver/Executor 负责并行计算 Spark-App。进一步, Driver 负责控制 Spark-App-BusinessLogic (如 Map→Reduce→Reduce) 整个过程, Executor 负责具体执行业务代码 (如 max)。在工作机制一节, 编者将给出 Spark-App 执行时, 其内部执行流。

2. ThirdPlatform 模式

既然 Client 可以向 Standalone 资源管理器提交 Spark-App, 那么只要做些适当的接口开发, 市面上的其他资源管理器也应当能够支持 Spark-App。事实的确是这样的, 和 Standalone 模式相比, ThirdPlatform 也能够完全支持 Spark-App, 只不过, 当 Spark-App 向 ThirdPlatform 请求资源时, ThirdPlatform 给予 Spark-App 分配的资源颗粒可能不够细致。

当前主流的集群资源管理器 YARN、Mesos。Mesos 是 Apache 使用 C++ 自主开发的一款高性能集群资源管理器, 其显著特征是执行效率高, 缺点是资源划分太细致, 参数较多不易控制。YARN 是 Hortonworks 开发的集群资源管理器, 有时也称分布式数据操作系统、分布式操作系统, 现 Hortonworks 已经捐赠给 Apache。

由于 Apache 和 Hortonworks 的大力推广, 加之 YARN 自身优异的性能, 目前 YARN 已成集群资源管理器方面事实上的标准。不但用户群广阔, YARN 上支持的组件也非常多。本节讲述的 ThirdPlatform 仅以 YARN 为例, 不讲述 Mesos。

1) 集群资源管理层

本书第五章已经讲述了 YARN, 故此处只给出最基本知识点。YARN 本身也采用 master/slave 架构, 其主服务就称为 ResourceManager, 从服务则称为 NodeManager。单机上的驻守进程 NodeManager 主要负责管理本机资源。集群资源管理者 ResourceManager 进程主要负责汇总各 NodeManager 进程汇报的单机资源情况 (图 9-13)。

实际上, 主服务 ResourceManager 还提供 Web-UI 功能、应用程序 (不仅仅是 Spark-App) 注册、任务调度、资源仲裁等功能。特别是 Spark-App 注册功能, 它是客户提交程序的入口。NodeManager 进程还提供启动和监管 SparkExecutor 进程功能。

2) 应用程序执行层

在集群上执行 Spark-App 时, 其执行过程依旧采用 master/slave 架构, 即 master 机负责控制程序整体执行流, slave 机负责具体执行某个任务 (由 master 分配给自己)。

Spark-App 执行层也采用 master/slave 模式, 只不过这里的主进程称为 SparkAppMaster, 从进程称为 SparkExecutor。SparkAppMaster 进程主要负责控制程序整个执行流 (图 9-14), 各个 SparkExecutor 负责并行执行某个具体任务 (由 SparkAppMaster 分配)。

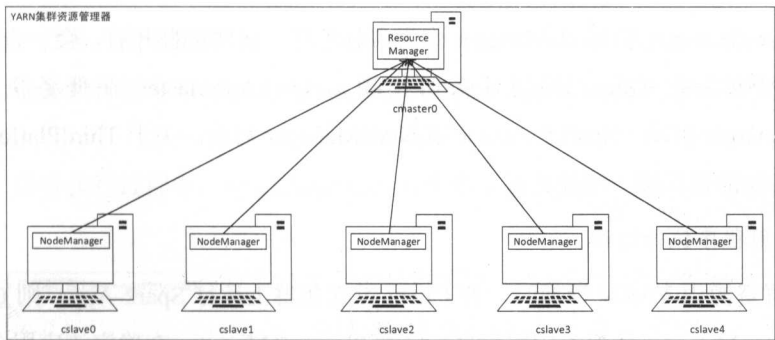


图 9-13 YARN 集群资源管理器

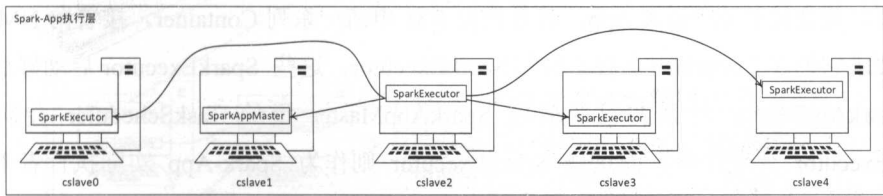


图 9-14 Spark-App 执行层

实际上，在 client 提交任务之前，YARN 集群中并不存在 SparkAppMaster 和 SparkExecutor，只存在 ResourceManager/NodeManager 进程。当 client 向 ResourceManager 提交任务后，ResourceManager 会先选中一个 Container 来执行 SparkAppMaster。待 SparkAppMaster 启动后，其会接管本 Spark-App，其首先向 RM 申请一系列 Container，接着向 NM 发出命令要求它们启动这些 Container。最后，SparkAppMaster 和 SparkExecutor 协作完成用户提交的 Spark-App。

3) 集群部署

当采用 ThirdPlatform(以 YARN 为例)模式时，所谓的部署 Spark，就是指部署 YARN 资源管理器和 Client，SparkAppMaster，SparkExecutor 和 SparkAppBusinessLogic 相关资源都在 Client 里。litteCstor 中 YARN 角色分配表见表 9-2。

表 9-2 litteCstor 中 YARN 角色分配表

角 色	主 机	进程名
资源管理主服务	cmaster0	ResourceManager
资源管理从服务	cslave0	NodeManager
	cslave1	
	cslave2	
	cslave3	
	cslave4	
客户端	iclient0	Client（非驻守）
	iclient1	

ResourceManager 和 NodeManager 作为常驻进程, 只要集群开启, 会一直运行下去; Client 进程则是在 Client 提交任务时出现, SparkAppMaster 在任务执行时, 由 ResourceManager 启动, SparkExecutor 由 NodeManager 启动。关于 ThirdPlatform 模式下的 Spark 实际部署步骤, 请参见下一小节。

4) 应用程序执行过程

当集群资源由 YARN 管理时, 若 Client 欲在集群上运行 Spark-App, 则 Client 须首先向 ResourceManager 注册本应用程序; 接着 ResourceManager 在确定该应用合法后, 会选定一 Container (YARN 资源分配基本单位) 执行 SparkAppMaster; 待 SparkAppMaster 启动后, 其会接管本 Spark-App, 其首先向 RM 申请一系列 Container, 接着向 NM 发出命令要求它们在 Container 上启动这些 SparkExecutor。这些 SparkExecutor 启动好后就会向 SparkAppMaster 注册自己; 最后 SparkAppMaster 里的 TaskScheduler 会向这些 SparkExecutor 分发任务, 而这些 SparkExecutor 则作为 Spark-App 实际执行者来执行 SparkAppMaster 分配给自己的任务并将结果返回给 SparkAppMaster。显然 Spark-App 是由 SparkAppMaster 和 SparkExecutor 协作完成的, 真正的并行化执行者是 SparkExecutor。

请读者务必注意, 每个 Spark-App 都包含一系列 SparkExecutor, 每个 SparkExecutor 则只属于一个 Spark-App。图 9-15 为正在运行两个 Spark-App 的 YARN 集群状态图, 从图中可以看出, 集群资源管理者为 ResourceManager 和 NodeManager, 两者之间为 master/slave 架构; Spark-App 执行者为 SparkAppMaster 和 SparkExecutor, 两者之间也为 master/slave 架构。iclient0 提交的 Spark-App0 拥有三个 SparkExecutor, iclient1 提交的 Spark-App1 则只有两个 SparkExecutor。

以 iclient0 提交的 Spark-App0 为例, 下面给出任务执行过程:

Step1 iclient0 向 ResourceManager 注册应用程序 (图中步骤①)。

Step2 ResourceManager 查看集群资源配置表, 选定一空闲 Container, 比如选中 cslave1 上的 Container (图中步骤②)。

Step3 ResourceManager 指示 cslave1 上的 NodeManager 在 Container 里启动 SparkAppMaster0 (图中步骤③)。

Step4 SparkAppMaster0 根据 SparkAppBusinessLogic (用户代码) 向 ResourceManager 申请一定数量的 Container (图中步骤④)。

Step5 SparkAppMaster0 指示 NodeManager 在这些 Container 上启动 SparkExecutor (图中步骤④)。

Step6 隶属于本 SparkAppMaster0 的 SparkExecutor 向本 SparkAppMaster 注册 (图中步骤⑤)。

Step7 SparkAppMaster0 根据用户编写的 SparkAppBusinessLogic 指挥 SparkExecutor

并行执行用户任务（图中步骤⑤）。

Step8 任务执行过程中，各 SparkExecutor 向 SparkAppMaster0 汇报任务执行进度（图中步骤⑤）。

Step9 任务执行过程中，SparkAppMaster0 向 SparkClient0 汇报任务执行进度（图中步骤⑥）。

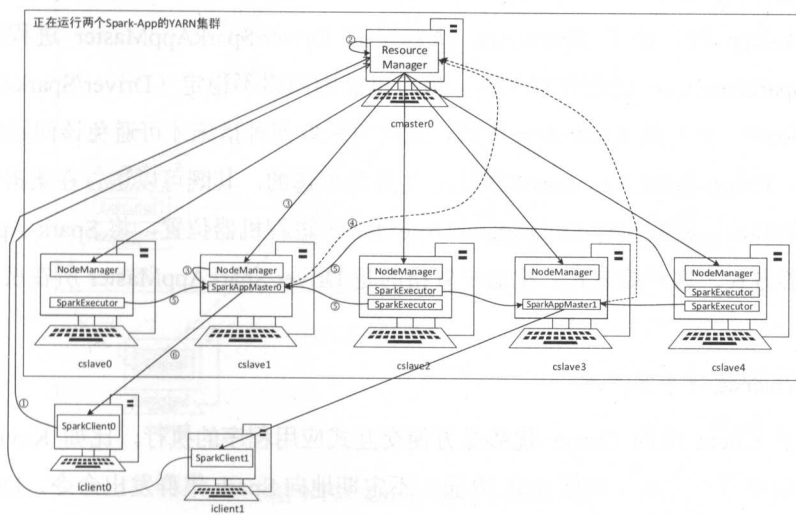


图 9-15 正在运行两个 Spark-App 的 YARN 集群

在 littleCstor 上执行这两个 Spark-App 时，SparkAppMaster0 名称就是 SparkAppMaster，并无“0”，此处为了便于说明才添加了“0”。

同理，在 iclient1 提交的 Spark-App1 时也主要包含这九步，由于标记太多，影响图清晰度，并未标出，请读者自行完成标注。

综上所述，在 ThirdPlatform 模式中（以 YARN 为例），当 Spark 集群中未运行任何 Spark-App 时，集群中仅存在 ResourceManager 和 NodeManager 进程。不过，当 client 向 YARN 集群（实际上是 ResourceManager 进程）提交 Spark-App 时（如 PI），集群中会出现多个 SparkExecutor 进程和一个 SparkAppMaster 进程。这两类进程（ResourceManager/NodeManager，SparkAppMaster/SparkExecutor）都采用 master/slave，其中 ResourceManager/NodeManager 负责资源管理，SparkAppMaster/SparkExecutor 负责并行执行 Spark-App。进一步，SparkAppMaster 负责控制 Spark-App-BusinessLogic（如 Map→Reduce→Reduce），SparkExecutor 负责具体执行业务代码（如取 max）。客户端进程 SparkClient 里包含 SparkAppMaster 代码、SparkExecutor 代码和 SparkAppBusinessLogic 代码（用户编写的业务逻辑代码）以及其他相关资源。当 Client（iclient0）需要向 YARN 集群提交任务时，只需要将这些资源提交给 YARN，YARN 会逐个启动这些计算进程。

3. 应用程序执行方式

对 Spark-App 的执行层，无论是 Standalone 模式下的 Driver/Executor，还是 ThirdPlatform 模式下的 SparkAppMaster/SparkExecutor，都是 master/slave 架构，问题是该 master 的物理机器在哪？比如 littleCstor 的物理机房处于英国，用户 Kevin 的 iclient7 位于中国南京，两者之间的网络有时不太稳定，当 Kevin 使用 iclient7 向远隔重洋的 littleCstor 提交 Spark-App 时，由于 Spark-App 执行层的 Driver/SparkAppMaster 进程要指挥各 Executor/SparkExecutor 进程并行工作，极有可能因网络不稳定（Driver/SparkAppMaster 运行在 iclient7）而导致 Spark-App 失败，此时可采取何种措施才可避免该问题呢？

其实，Driver/SparkAppMaster 的运行位置是可选的，其既可以运行在集群外，也可以运行在集群内。根据执行 Driver/SparkAppMaster 进程机器位置，将 Spark-App 执行方式分为集群方式、客户端方式。机器位置指的是 Driver/SparkAppMaster 所在机是否位于集群内。

1) Driver 运行于客户端

运行于 Client 端的 Driver 优势是方便交互式应用程序的执行，比如 Kevin 可以在 iclient0 上以交互式方式（类似 SQL 界面）不定期地向 Spark 集群发出命令，Driver 收到 Kevin 给出的指令就指挥 Spark 集群工作，没收到就一直等待。显然，交互式方式能够大大方便数据分析师分析数据。此外，当 Driver 运行于客户端时，须确保该机和集群各节点处于同一网络。

图 9-16 为 Driver 运行于客户端时示意图，9.3 节编者将给出以交互式方式操作 Spark 集群示例。

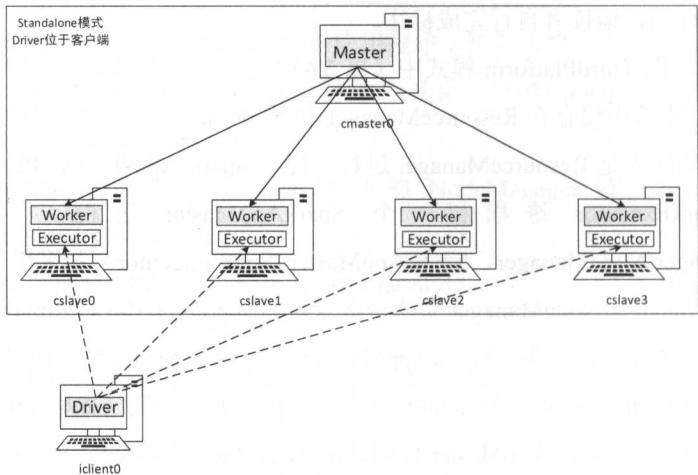


图 9-16 Driver 运行在客户端

2) Driver 运行于集群

当 Driver 运行于集群时，客户端的 SparkClient 依旧存在，不过该进程的功能退化到显示 Spark-App 执行进度，无法通过该方式以交互式方式运行 Spark-App。

图 9-17 为 Driver 运行于集群时的示意图，该方式一般用于批量处理 Spark-App。

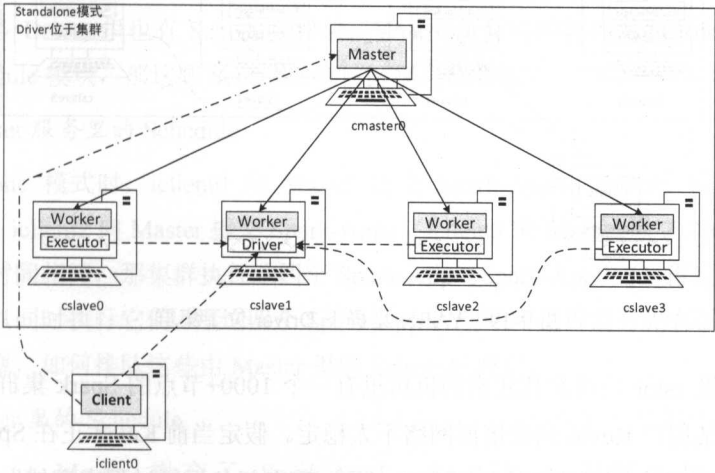


图 9-17 Driver 运行于集群

3) YARN 集群中 SparkAppMaster 运行方式

当 Spark 使用 YARN 集群资源管理器时，SparkAppMaster 也支持集群内运行和集群外运行两种方式，图 9-18 和图 9-19 为这两种方式示意图，两种方式的优缺点和 Standalone 模式下相同。

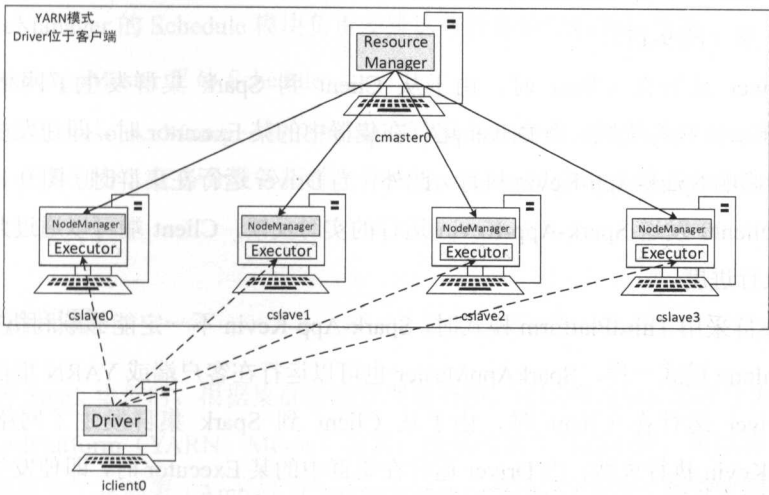


图 9-18 YARN 集群下 Driver 位于客户端

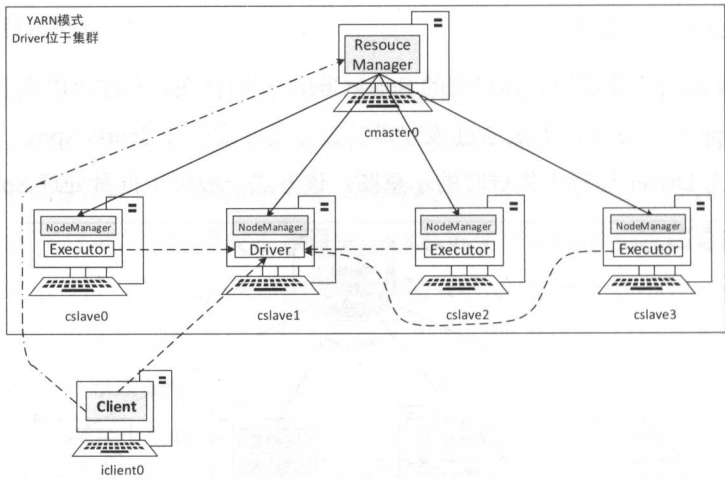


图 9-19 YARN 集群下 Driver 位于集群

例 1 假设 cstor 公司在其北京的机房里有一个 1000+节点的 Spark 集群 (Standalone 模式), 南京某用户 Kevin 到此集群网络不太稳定。假定当前 Kevin 正在 Spark 集群上运行一个 Spark 任务 (不妨称 Spark-App-Kevin), 若在任务运行过程中突然发生网络中断, 试问该任务是否能够成功完成, 若不能, 可采取何机制确保任务顺利执行。当集群采用 ThirdPlatform 模式时情况又如何。

解① 不一定。在 Standalone 模式下, Driver 进程控制附属于本 Driver 的所有 Executor, 它需要经常和各个 Executor 进行 TCP 通信, 故必须确保 Driver 进程所在机器和 Spark 集群在同一网络且连接稳定。

但是, Driver 不仅可以运行在远离集群的 Client 机 (图 9-16), 也可以运行在集群中某 Executor 里 (图 9-17)。

当 Driver 运行在 Client 时, 由于从 Client 到 Spark 集群发生了网络中断, 故 Spark-App-Kevin 执行失败; 当 Driver 运行在集群中的某 Executor 时, 即使发生了网络中断, 也不会影响 Spark-App-Kevin 执行。此外, 当 Driver 运行在集群时 (图 9-17), Driver 进程会向 iclient0 发送 Spark-App-Kevin 运行的实时反馈, Client 端可以通过此反馈得知任务当前执行进度。

②当集群采用 ThirdPlatform 模式时, Spark-App-Kevin 不一定能够顺利执行。这是因为同 Standalone 模式一样, SparkAppMaster 也可以运行在客户端或 YARN 集群里。

当 Driver 运行在 Client 时, 由于从 Client 到 Spark 集群发生了网络中断, 故 Spark-App-Kevin 执行失败; 当 Driver 运行在集群中的某 Executor 时, 即使发生了网络中断, 也不会影响 Spark-App-Kevin 执行。同 ThirdPlatform, 当 Driver 运行在集群时 (图 9-19), Client 会收到 Driver 进程发送的任务执行进度信息。图 9-18 和图 9-19 中, 编者并

未将 Driver 改为 SparkAppMaster，这是因为在真实 YARN 集群上，依旧称为 ApplicationMaster，重点理解其体系架构，读者不必纠结进程名。

4. Schedule

Standalone 模式下，集群资源管理器主服务 Master 里有 Schedule 模块；Spark-App 执行器主服务 Driver 里也有 Schedule 模块；YARN 集群资源管理器的 ResourceManager 里也有 Schedule 模块，那这些 Schedule 有何区别与联系呢？

1) Master 服务里的 Schedule

Standalone 模式时，iclient0 向 Master 提交 Spark-App0，iclient1 向 Master 提交 Spark-App1，iclient2 向 Master 提交 Spark-App2，iclient3 向 Master 提交 Spark-App3。假定它们提交时间相同，那集群执行这四个 Spark-App（Spark-App0~3）肯定有先后顺序，即使集群可以同时执行它们，那这四个 Spark-App 启动顺序也肯定有先有后^[1]。到底谁先执行，谁阻塞，如何排队这些由 Master 里的 Schedule 决定。

2) Driver 里的 Schedule

iclient0 向 Master 提交了 Spark-App1，Spark-App1 里任务是最终规约为 Map→Reduce0→Reduce1，则这个过程由 Driver 里的 Schedule 控制，至于 Reduce0 里具体做了事（如取 max），则由各个 Executor 负责，该 Schedule 的工作就是为这些 Executor 分配任务。

3) ResourceManager 里的 Schedule

YARN 模式时，iclient0 向 ResourceManager 提交了 Spark-App0，iclient1 向 ResourceManager 提交了 Distributed-Shell，iclient2 向 ResourceManager 提交了 Pig-App。集群资源管理主服务 ResourceManager 的 Schedule 模块负责安排这些任务的先后执行顺序。

4) SparkAppMaster 里的 Schedule

iclient0 向 ResourceManager 提交的 Spark-App0 任务最终规约为“Map→Reduce0→Reduce1”，任务划分、任务分发等指挥工作由 SparkAppMaster 里的 Schedule 负责。

9.1.3 集群部署

在部署 Spark 集群时，根据集群资源管理器类型，可以将 Spark 部署分为 Standalone 部署和 ThirdPlatform（YARN、Mesos）部署；根据部署工具的不同，可以将 Spark 部署分为手工部署和工具部署（Ambari、Cloudmanager），图 9-20 为这几种方式的组合，从图中可以看出，Spark 部署方式较多。

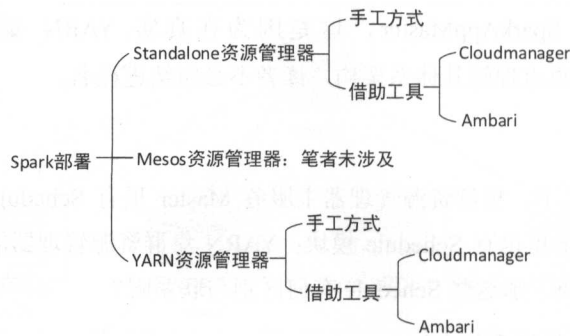


图 9-20 Spark 部署方式

不过，为保持 littleCstor 完整性，编者建议使用 Ambari 将 Spark 部署 YARN 集群上；为充分挖掘 Spark 处理能力，编者建议使用 ClouderaManager 以 Standalone 模式部署 Spark 集群。由于本书写作期间，Ambari 只支持 Spark-YARN，不支持 Spark-Standalone，ClouderaManager 支持 Spark-Standalone 模式，不支持 Spark-YARN，为保持整个集群完整性，编者分别选用了 Ambari 部署 Spark-YARN 和手工方式部署 Spark-Standalone。

1. 独立模式（Standalone）

Spark 部署步骤非常简单，将其解压，然后填写 slaves 文件即可。不过，在实战中，还要配置 Spark 指向 HDFS，设置 Master 和各 Worker 内存，CPU 上下限。

1) 最精简部署

当 Spark 选用 Standalone 资源管理模式时，须将 Master 部署到 master 机，Worker 部署到所有 slave 机，client 部署到所有 client 机。启动集群时，master 机启动 Master 进程，所有 slave 机启动 Worker 进程。运行 Spark-App 任务时，client 机启动 Driver 进程向 Master 提交任务，slave 机启动 Executor 执行任务。在集群上以 Standalone 方式部署 Spark 时，主要包含如下九大步骤。

Step1 制定部署规划。

Step2 准备硬件机器，准备机器操作系统环境，准备机器网络环境。

Step3 对集群内每一台机器，修改机器名，关闭防火墙，安装 jdk。

Step4 为每台机器，添加集群级别域名映射。

Step5 打通主节点到自身无密钥认证，打通主节点到所有从节点无密钥认证。

Step6 主节点解压 Spark，配置 spark。

Step7 将配置好的 Spark 复制至所有 slave 机。

Step8 启动 Spark。

Step9 测试 Spark。

显然，在上述实际部署过程中，各机并未分角色部署不同程序包，而是所有机器都部署相同的 Spark 包。不过，在启动 Spark 时，各机会根据配置文件，启动不同角色的进

程。在讲述 Spark 部署之前，编者再次讲述在集群管理中至关重要的“ssh”。

在集群环境下，经常有需求需要从中心机上执行 Shell 脚本，操作集群内所有机器。这类操作一般都借助 ssh 完成。由于 ssh 非常重要，编者再次以例题方式讲述如何实现 ssh 无密钥认证。

例 2 请以 `allen` 用户实现从 `cmaster0` 机到 `cslave0~cslave4` 和 `cmaster0` 自身无密钥登录。

解 实现 cmaster0 到自身和其他 slave 机无密钥认证，首先需要 cmaster0 机（以 allen 用户）生成公私钥。当需要实现 cmaster0 到 cmaster0 自身无密钥登录时，只要将上一步生成的公钥追加到文件 authorized_keys 里；当需要实现 cmaster0 到 cs slave0 无密钥登录时，只要将 cmaster0 的公钥追加到 cs slave0 的认证文件 authorized_keys 里即可；当需要实现 cmaster0 到 cs slave1 无密钥登录时，只要将 cmaster0 的公钥追加到 cs slave1 的认证文件 authorized_keys 里即可；同理，当需要实现 cmaster0 到所有 slave 机无密钥认证时，只要将 cmaster0 的公钥拷贝至各 slave 机的认证文件 authorized keys 里即可。

Step1 cmaster0 使用命令“ssh-keygen”生成公私钥，执行过程如图 9-21 所示。

```
[allen@master0 ~]$ ssh-keygen ————— 生成公私密钥对命令
Generating public/private rsa key pair.
Enter file in which to save the key (/home/allen/.ssh/id_rsa):
Created directory '/home/allen/.ssh'.
Enter passphrase (empty for no passphrase): ←————— 直接回车
Enter same passphrase again: ←————— 再次直接回车
Your identification has been saved in /home/allen/.ssh/id_rsa.
Your public key has been saved in /home/allen/.ssh/id_rsa.pub.
The key fingerprint is:
ec:de:da:d2:91:e6:e8:cd:02:95:f8:e3:e6:d3:6e:75 allen@master0
The key's randomart image is:
+--[ RSA 2048 ]--+
|
|. . .
| . o +
| o +S.
| +.o . E
| +.=.o
| *. *o
| o.=+oo
+-----+
[allen@master0 ~]$
```

图 9-21 cmaster0 机 allen 用户生成 ssh 密钥

Step2 将 cmaster0 的公钥拷贝至本机的 “/home/allen/.ssh/authorized_keys” 文件里, 以实现 cmaster0 到 cmaster0 无密钥登录。

```
[allen@cmaster0 ~]$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys #cmaster0,allen 用户,实现本机登录
```

Step3 将 cmaster0 公钥拷贝并追加到所有 slave 的 “/home/allen/.ssh/authorized_keys” 文件里, 以实现 cmaster0 到 cslaveX 无密钥登录, 比如下命令将 cmaster0 公钥远程拷贝至 cslave0 机 “/home/allen/” 目录下。

```
[allen@cmaster0 ~]$ scp ~/.ssh/id_rsa.pub cslave0:~/ #allen 将 cmaster0 公钥拷至 cslave0
```

接着在 `cslave0` 上，将刚拷过来的文件追加到本机认证文件 `authorized_keys` 里，即可实现 `cmaster0` 到 `cslave0` 无密钥认证。

```
[allen@cslave0 ~]$ mkdir .ssh #cslave0,allen,不存在.ssh文件夹时,须新建
[allen@cslave0 ~]$ cat id_rsa.pub >> ~/.ssh/authorized_keys #allen 将公钥追加到本机认证文件
```

上述操作完成后,请读者自行参照 Step3,自行实现 cmaster0 到其他所有 slave 无密钥认证。

Step4 测试从 cmaster0 到 cmaster0 自身和所有 slave 无密钥认证,比如下述命令即是验证 cmaster0 登录到其他机器时是否还需要密码。

```
[allen@cmaster0 ~]$ ssh cmaster0 #allen,测试从 cmaster0 无密钥登录 cmaster0
[allen@cmaster0 ~]$ ssh cslave0 #allen,测试从 cmaster0 无密钥登录 cslave0
```

请读者自行将“cslave0”换成“cslave1”,“cslave2”...,完成所有认证。实际上,编者已在第二章和第三章讲述过无密钥认证,这里编者再次讲述,可见 ssh 在集群管理中的重要性。在集群环境下,当需要实现从一台机器登录其他机器时,当需要从一台机器管理其他所有机器时,都需要借助 ssh。不过对于 Hadoop 和 Spark 来说,ssh 实质上是不必要的,其不参与任何程序执行,只在集群启动脚本里被调用。

有了“ssh”这一背景知识,下面编者给出一个现实场景,接着编者在此场景下逐步实现 Spark 部署。

场景

现有一堆机器 cmaster0, cslave0~N, iclient0、iclient1,试在这些机器上以 Standalone 模式部署 Spark 集群。

Step1 制定部署规划。

根据机器名,编者做出的规划为: cmaster0 机充当主节点,部署 Master 服务; cslaveX 充当从节点,部署从属服务 Worker 进程; iclient0 和 iclient1 充当客户节点,部署客户端,图 9-22 为部署规划的效果图。

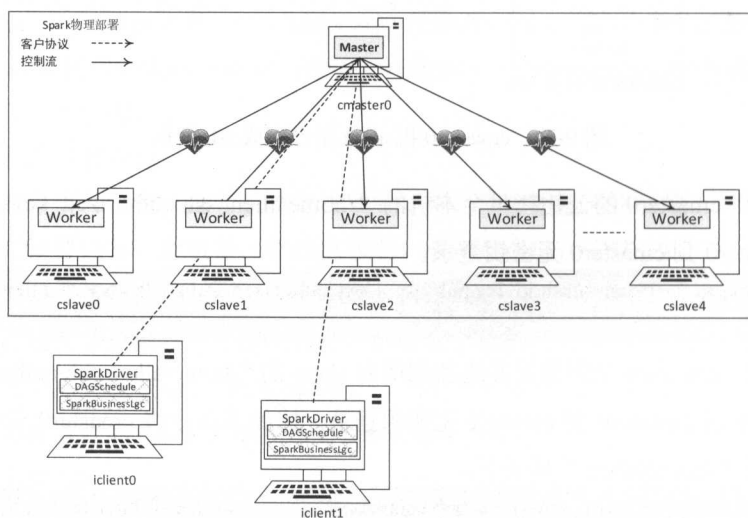


图 9-22 独立模式的 Spark 集群效果图

Step2、3、4、5:

在制定好上述部署规划后，请读者自行准备硬件机器，各台机器的操作系统统一为 CentOS-6.7-x86_64，除了默认的 root 用户外，统一添加 allen 用户，编者将以 allen 用户来安装 Spark。在 Spark 部署之前，需要对集群中每台机器，修改其机器名，添加域名映射，关闭防火墙，安装 jdk，这些操作命令编者已经在第 3 章讲述，故不再赘述。

Step6 主节点解压 spark，配置 spark。

One 下载 Spark，将 spark 拷至 cmaster0 机。

在下载 Spark 时，请读者到官网下载最新的稳定版 Spark，比如编者下载的就为当前最新稳定版 “spark-1.5.2-bin-hadoop2.6.tgz”。

Two 在 cmaster0 上，使用如下命令，解压 “spark-1.5.2-bin-hadoop2.6.tgz”。

```
[allen@cmaster0 ~]$ tar -zxvf spark-1.5.2-bin-Hadoop2.6.tgz ~/
```

Three 编辑 Spark 的 slaves 文件，将下述内容添加到 slaves 文件里。

```
cslave0
cslave1
cslave2
cslave3
```

上述内容表示当前的 Spark 集群共有四台 slave 机，这四台机器的机器名分别是 cslave0~3。通过下述 vim 命令编辑 Spark 的 slave，将上述内容添加到配置文件里即可：

```
[allen@cmaster0 ~]$ vim spark-1.5.0-bin-hadoop2.6/conf/slaves
```

编辑文件后，读者可使用如下 cat 命令查看 slaves 文件里内容是否添加成功，比如编者的 slaves 文件如图 9-23 所示。

```
[allen@cmaster0 spark-1.5.2-bin-hadoop2.6]$ cat conf/slaves
# A Spark Worker will be started on each of the machines†listed below.
cslave0
cslave1
cslave2 ← 当前Spark集群所有从节点（不包括主）
cslave3
[allen@cmaster0 spark-1.5.2-bin-hadoop2.6]$
```

从节点列表文件

图 9-23 slaves 文件效果图

Spark 的所有配置就这一个，无需其他任何配置（指定 Spark 使用 Hadoop 集群除外），无须指定主节点是哪台机器，这是因为 Spark 默认认为启动 “start-all.sh”的那台机器就是集群主节点。

Step7 将配置好的 Spark 复制至所有 slave 机。

将配置好的 spark-1.5.2-bin-Hadoop2.6 拷贝至 cslave0~cslave3，iclient0~iclient1。下面的命令可实现将 spark-1.5.2-bin-Hadoop2.6 同时拷至 4 台 slave。

```
[allen@cmaster0 ~]$ for x in `cat spark-1.5.2-bin-hadoop2.6/conf/slaves`;do echo $x ;scp -r spark-1.5.2-bin-Hadoop2.6 allen@$x:~/;done;
```


请读者参考此循环命令, 实现将 spark-1.5.2-bin-hadoop2.6 拷贝至 iclient0 和 iclient1。

Step8 启动 Spark 集群。

```
[allen@cmaster0 ~]$ spark-1.5.2-bin-hadoop2.6/sbin/start-all.sh
```

此命令只能在 cmaster0 上执行, 不可以在其他 slave 机或客户机上执行, 脚本默认本机即为 Spark 主节点。和 start 命令类似, 用户可以使用 stop-all.sh 命令关闭整个 Spark 集群。

Step9 验证 Spark 是否成功启动。

One 验证进程。

在 Spark 集群启动后, 用户可以通过在 cmaster0 和 cslave0~3 上分别执行 jps 命令查看看到对应进程, 即 cmaster0 进程为 Master, slave 机进程为 Worker。

Two 验证 WebUI。

此外, 用户还可在 FireFox 浏览器里输入地址 <http://cmaster0:8080>, 即可看到 Spark 的 Web UI。此页面上包含了 Spark 集群主节点、从节点等各类统计信息。

Three 验证提交 Spark-App

当用户拥有了自己的一个 Spark 集群后, 最想做的事可能就是向 Spark 集群提交 Spark-App 了。Spark-App 的经典实例是计算圆周率 PI, 下面的 Shell 命令完成在 iclient0 机上, 以 allen 用户向 Spark 集群的 Master 服务 (spark://cmaster0://7077) 提交 Spark-PI。集群中各个 slave 机会启动 Executor 进程来执行此任务。

```
[allen@iclient0 spark-1.5.2-bin-hadoop2.6]$ bin/spark-submit --master spark://cmaster0:7077 \
> --class org.apache.spark.examples.SparkPi lib/spark-examples-1.5.2-hadoop2.6.0.jar
```

上述命令可以执行的前提是集群已部署并启动了 Spark 服务, 在 Spark-PI 执行过程中, 读者可在 FireFox 浏览器里打开 “<http://cmaster0:8080>” 和 “<http://cmaster0:4040>” 查看任务执行进度; 可在 iclient0 上输入 jps 查看 Spark 客户端 Driver 进程, 在任意一个 slave 节点上输入 jps 查看 Executor 进程。

2) 配置 Spark 使用 HDFS

精简方式部署的 Spark 集群须进行配置才可存取 HDFS 文件, 设置 Spark 使用 HDFS 非常简单, 在 “/home/allen/spark-1.5.2-bin-Hadoop2.6/conf/spark-env.sh” 引入 Hadoop 配置文件即可, 操作步骤如下。

Step1 若 Spark 集群正在运行, 请关闭集群。

Step2 获取 Hadoop 集群配置文件位置。

手工部署的 Hadoop 集群配置文件一般在目录 “Hadoop 解压目录/etc/hadoop” 下, Ambari 部署的 Hadoop 集群配置文件一般在 “/etc/hadoop” 下。本书选择使用 Ambari 部署的 Hadoop 集群。

Step3 编辑 cmaster0 机上的“/home/allen/spark-1.5.2-bin-Hadoop2.6/conf/spark-env.sh”文件，将下列内容追加到文件中：

```
export HADOOP_CONF_DIR=/etc/hadoop
```

Step4 在集群中所有 slave (cslave0~4) 机器和 client (iclient0~1) 机上，参考步骤 2，编辑这些机器 spark-env.sh 文件，将上述内容追加到文件中。

Step5 启动 Spark 集群。

只配置这一个参数，即可实现 Spark 操作 HDFS 数据。在步骤 4 中，读者也可采用 scp 命令将配置好的文件远程拷贝到其他机器。

显然，当你拥有了自己的一个 Spark 集群且此集群持久化层指向 HDFS 时，你最想做的事应该就是向 Spark 集群提交一个使用 HDFS 的 Spark-App 了，下面进行操作。

除了 PI，Spark 上另一个经典应用就是 SparkWordCount，现在，编者向 Spark 提交 SparkWordCount，该程序计算的文件为 HDFS 上的 note2.txt，文件在 HDFS 中完整路径为“hdfs://cmaster0:8020/user/allen/notes/note2.txt”，文件内容用户自定，文件大小用户自定。

Step1 上传 note2.txt 文件。

该步请读者自行完成，上传时请务必注意使用“allen”用户，文件在 HDFS 中的绝对路径。

Step2 使用 Shell 命令向集群提交 Spark-App。

```
[allen@iclient0 spark-1.5.2-bin-hadoop2.6]$ bin/spark-submit --master spark://cmaster0:7077 \
> --class org.apache.spark.examples.JavaWordCount \
> lib/spark-examples-1.5.2-hadoop2.6.0.jar /user/allen/in/note2.txt
```

上面的命令实现调用 WordCount 程序，计算 HDFS 上的文件“/user/allen/in/note2.txt”，计算结果直接输出到控制台。

显然，在配置 Spark 使用 HDFS 后，若需要 Spark-App 计算 HDFS 上文件，在程序里指明 HDFS 上文件路径即可，无需其他配置。

在 Spark-WordCount 执行过程中，读者可在 FireFox 浏览器里打开 <http://cmaster0:8080> 和 <http://cmaster0:4040> 查看任务执行进度；可在 iclient0 上输入 jps 查看 Spark 客户端 Driver 进程，在任意一个 slave 节点上输入 jps 查看 Executor 进程。

2. 宿主于第三方平台（YARN）

就像可以把 MapReduce 部署到 YARN 上一样，也可以把 Spark 部署到 YARN 集群。不过对于 YARN 集群来说，严格地说，Spark 服务并无“部署”这一说法，和 MapReduce、DistributedShell 一样，实质上它只是 YARN 的一个客户端，只需要客户端持有 Spark 相关资源（Jar、程序包、命令包等各类资源），使用时直接向 YARN 集群提交这类资源即

可。YARN 集群会使用这些代码资源，在 YARN 集群内启动相关实体执行 Spark-App。

正如图 9-24 所示，Spark 执行之前，YARN 集群中并不存在任何 Spark 痕迹，没有 Spark 进程，没有 Spark Jar 包，甚至 YARN 连 Spark 是谁都不知道。此时若客户端需要提交 Spark-App，那么此客户端需要持有 Spark（SparkAppMaster）和 Spark 用户层面的程序代码（SparkBusinessLogic）资源。首先，Client 使用相关协议向 ResourceManager 申请第一个 Container 来执行 Spark 的主服务 SparkAppMaster；待 SparkAppMaster 启动后，其会接管本 Spark-App，接着主服务 SparkAppMaster 向 ResourceManager 申请一定数量的 Container 来执行 Spark 从属进程 SparkExecutor；最后这些 SparkExecutor 在 SparkAppMaster 统一指挥下，并行处理用户层面的 SparkBusinessLogic 里具体任务（图 9-15），这部分内容已在上节讲述，有疑问的读者请再次阅读上一节。

YARN 部署也分手工和工具两种方式，为保持 littleCstor 完整性，编者直接使用第 3 章已部署好的 YARN。

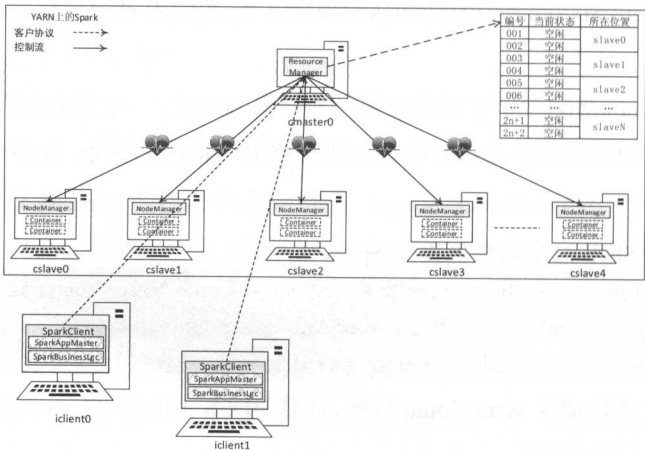


图 9-24 Spark 部署于 YARN 集群

使用 Ambari 部署 Spark 可以说是一键操作，难点几乎都在 Ambari 工具本身部署上，以下步骤从无到有，简单介绍了 Ambari 自身部署和使用 Ambari 部署 Spark 的大概步骤：

- Step1 制定部署规划。
- Step2 准备硬件机器和 OS 环境。
- Step3 配置单机 OS 环境和集群环境。
- Step4 部署 Ambari-server。
- Step5 使用 Ambari-server 部署 HDFS、YARN、Spark。

例 3 使用 Ambari 为 littleCstor 部署 Spark。

解 由于大数据平台涉及太多组件，故部署之前最好制定一个完备的部署计划，否则

极有可能导致由于各机角色分配混乱而部署失败。

本题以第 3 章为前提，只给出 Spark 部署规划表（表 9-3）和部署效果图（图 9-24），具体部署过程请参见第 3 章。读者须注意，图中 iclient0 到 cmaster0 或 cslave0、2 的链接实际上并不存在，也就是 iclient0 并不需要向任何机器汇报心跳包，只有当 iclient0 需要向集群提交 spark 任务时，它才会主动连接 cmaster0。

表 9-3 littleCstor 上 Spark 部署规划

机 器	角 色	部署服务
cmaster0	Spark 客户端	Spark 客户端 Shell 接口

正如图 9-24 所示，在 YARN 集群中，和 MapReduce、Pig、Hive 等组件一样，Spark 只是 YARN 的一个客户端。

9.1.4 计算模型

Spark 是一个用于并行处理海量数据的分布式计算框架，其内部对任务的并行策略是 M-S-R 范式。一个标准的 Spark-App 可拆分成一系列 M-S-R 任务，Spark 内置了 DAGSchedule，直接将 DAG 型 M-S-R 任务调度到集群上执行。Spark 的计算模型是，对分散于集群内各个计算节点的数据：

- ①如果需要对这些数据进行若干次本地计算那就进行若干次本地计算（Map）。
- ②如果需要将不同机器上的数据进行若干次合并那就进行若干次网络合并（Shuffle）。
- ③对于 Shuffle 后数据，如果还需要进行若干次 Map 或 Shuffle 操作，那就进行若干次 Map 或 Shuffle 操作。
- ④可按业务需求，可对上述①、②、③进行任意组合，直接任务结束。

在实际开发过程中，伯克利设计人员将上述过程设计得非常巧妙，比如使用 Partition 对应于存于在不同机器上的数据，使用 RDD 将多个 Partition 组合到一起（这样可以对若干个 Partition 进行统一操作），在实战 RDD 环节将重点讲述这些内容，下面先讲述 Spark 计算模型。

1. 计算模型引例

下面给出几个场景来说明 Spark 计算模型。

1) 场景 1: 单输入源

现有一原始文件 rawFile0（图 9-25），要求将 rawFile0 均匀加载到 cslave0、cslave1

和 cslave2 中, 然后对 rawFile0 中数据执行过滤操作, 过滤时要求只保留"aa"、"bb"这两种字符。接着, 将"aa"全部发往 cslave0 上、"bb"全部发往 cslave1 上。最后统计"aa"和"bb"这两个单词出现次数。

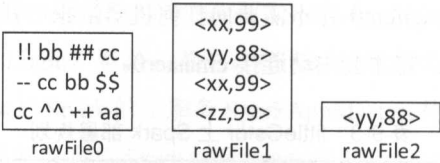


图 9-25 原始数据

下面给出该场景 Spark-App 代码和程序执行流程图。

(1) 程序代码

下述 Spark 代码可完成上述要求, 和题中要求不符的是, 在将"aa"发往 cslave0 之间, 编者已经对"aa"进行了置 1 操作, 不过这并不影响程序结果。

```
val conf = new SparkConf()
val sc: SparkContext = new SparkContext(conf)
val rawRDDA=sc.parallelize(List("!! bb ## cc","%% cc bb %%","cc && ++ aa"),3)
var tmpRDDA1=rawRDDA.flatMap(line=>line.split(" "))
var tmpRDDA2=tmpRDDA1.filter(allWord=>{allWord.contains("aa") || allWord.contains("bb")})
var tmpRDDA3=tmpRDDA2.map(word => (word, 1))
import org.apache.spark.HashPartitioner
var tmpRDDA4=tmpRDDA3.partitionBy(new HashPartitioner(2)).groupByKey()
var tmpResultRDDA=tmpRDDA4.map((P: (String, Iterable[Int])) => (P._1, P._2.sum))
```

(2) 程序执行流程图

上述代码可绘制成如下执行流程图 (图 9-26), 代码中的 sc.parallelize(3)就是将数据并行加载到三台机器上, partitionBy(new HashPartitioner(2)).groupByKey 就是将之前的 3 台机器 Shuffle 成两台机器。

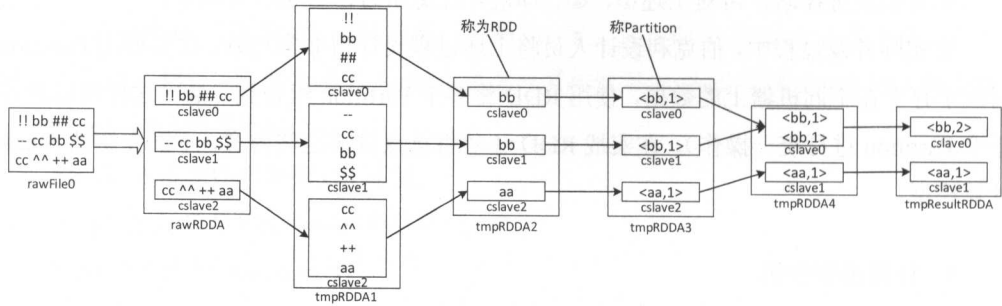


图 9-26 Spark-WordCount 执行流程图

在 Spark 中, 称某机上的一个固定数据块为一个 **Partition**; 称一系列相关 Partition

组合为一个 RDD, 如 tmpRDDA2 拥有三个 Partition 而 tmpResultRDDA 拥有两个 Partition。通过上述代码及其执行流程图可以看出:

①RDD 是数据统一操作所在地。

代码中任意一个操作 (如 flatMap、filter、map), RDD 内的所有 Partition 都会执行。如在 rawRDDA→tmpRDDA1 的转换过程中, 在 rawRDDA 上执行“flatMap(line=>line.split(" "))”时, 隶属于 rawRDDA 的三个 Partition (分别为: cslave0 上的“!! bb ## cc”, cslave1 上的“-- cc bb \$\$”和 cslave2 上的“cc ^^ ++ aa”)都要执行 flatMap 操作。

②RDD 是数据并行化所在地。

既然隶属于本 RDD 的所有 Partition 都要执行相同操作, 那么只要这些 Partition 存于不同机器, 执行时就由不同机器同时执行, 也就是并行执行。如在“tmpRDDA1→tmpRDDA2”的转换过程中, 在“tmpRDDA1.filter(...)”操作时, cslave0、cslave1、cslave2 都在同时处理存于本机的 Partition, 即多机并行处理。

准确地说, 图中所有 RDD 转换都是并行的, 包括从“tmpRDDA3→tmpRDDA4”的 Shuffle 转换, 都是多机并行处理。

③RDD 并行化范式主要有 Map 范式和 Shuffle 范式。

Map 范式指的是只对本 Partition 上数据进行操作, 其操纵的数据对象不跨越多个 Partition, 即不跨越网络。

Shuffle 范式指的是对不同 Partition 上的数据进行重组, 其操纵的数据对象不跨越多个甚至是所有 Partition, 即跨越网络。

显然, 图 9-26 中的 Map 范式为: rawRDDA→tmpRDDA1、tmpRDDA1→tmpRDDA2、tmpRDDA2→tmpRDDA3、tmpRDDA4→tmpResultRDDA。属于 Shuffle 范式的转换则只有一个: tmpRDDA3→tmpRDDA4。

④RDD 是数据隔离核心。

明显, rawRDDA 和 tmpRDDA1 是不同的 RDD。通过 RDD, 可实现数据完美隔离。

⑤RDD 是数据转换核心。

称从本 RDD 到下一个 RDD 为一个 RDD 转换。

显然, 上述过程发生了大量 RDD 转换。

2) 场景 2: 多输入源

现有两个原始文件 rawFile1 和 rawFile2, 要求将 rawFile1 的内容均匀加载到 cslave3、cslave4 上, 接着对 rawFile1 的数据进行去重操作, 即 rawFile1 里每个条目只保留一项; 要求将 rawFile1 内存加载到 cslave5, 接着对第一问的结果数据, 再去除 rawFile0 中所含条目。

下面给出能够完成上述任务的 Spark-App 和程序执行流程图。

(1) Spark-App 代码

根据要求，现编写如下 Spark 代码，代码中的“sc.parallelize(2)”就是将数据并行加载到两台机器上，同理“sc.parallelize(1)”是将数据加载到一台机器上。

```
val conf = new SparkConf()
val sc: SparkContext = new SparkContext(conf)
val rawRDDB=sc.parallelize(List(("xx", 99), ("yy", 88),("xx", 99), ("zz", 99)), 2)
val rawRDDC=sc.parallelize(List(("yy", 88)), 1)
var tmpResultRDDBC=rawRDDB.distinct.subtract(rawRDDC)
```

(2) 程序执行流程图

上述代码可绘制成执行流程图（图 9-27）。

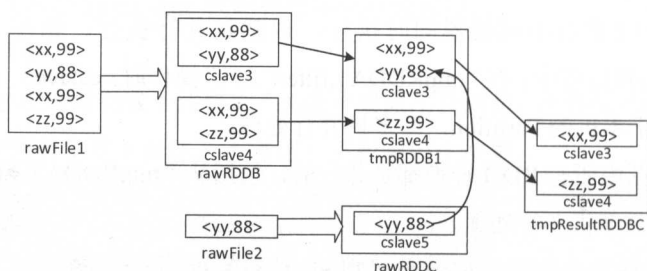


图 9-27 Spark-两个输入 RDD 结合过程

显然，容易看出，对于 Spark 来说：

⑥在同一个操作中，可以有不同的输入源。

显然，代码中的 subtract()就是两个 RDD 相减，而这两个又是来自两个不同的输入文件。

3) 场景 3: 复杂情况

现有三个原始文件 rawFile0、rawFile1、rawFile2，要求统计 rawFile0 中“aa”、“bb”两个单词出现次数；要求对去重后的 rawFile1 文件，再去掉 rawFile2 中内容；最后，要求将上述两个结果合并成同一个文件并存入 HDFS。

下面给出能够完成上述任务的 Spark-App 和程序执行流程图。

(1) Spark-App 代码

显然，将“场景 1”和“场景 2”的代码结合，即可完成比较复杂的处理机制，具体编写时，程序代码如下：

```
val conf = new SparkConf()
val sc: SparkContext = new SparkContext(conf)
val rawRDDA=sc.parallelize(List("!! bb ## cc","%% cc bb %%", "cc && ++ aa"),3)
val rawRDDB=sc.parallelize(List(("xx", 99), ("yy", 88),("xx", 99), ("zz", 99)), 2)
val rawRDDC=sc.parallelize(List(("yy", 88)), 1)
import org.apache.spark.HashPartitioner
```



```
var tmpResultRDDA=rawRDDA.flatMap(line=>line.split(" ")).filter(allWord=>{allWord.contains("aa")
|| allWord.contains("bb")}).map(word => (word, 1)).partitionBy(new HashPartitioner(2)).groupByKey().map((P:
(String, Iterable[Int])) => (P._1, P._2.sum))
var tmpResultRDDBC=rawRDDDB.distinct.subtract(rawRDDDC)
val resultRDDABC= tmpResultRDDA.union(tmpResultRDDBC)
resultRDDABC.saveAsTextFile("HDFS 路径")
```

(2) 程序执行流程图

根据上述代码和程序要求，编者给出程序执行流程图（图 9-28），从图中可明显看出附属于某 RDD 的所有 Partition 都在多机上并行地执行着。

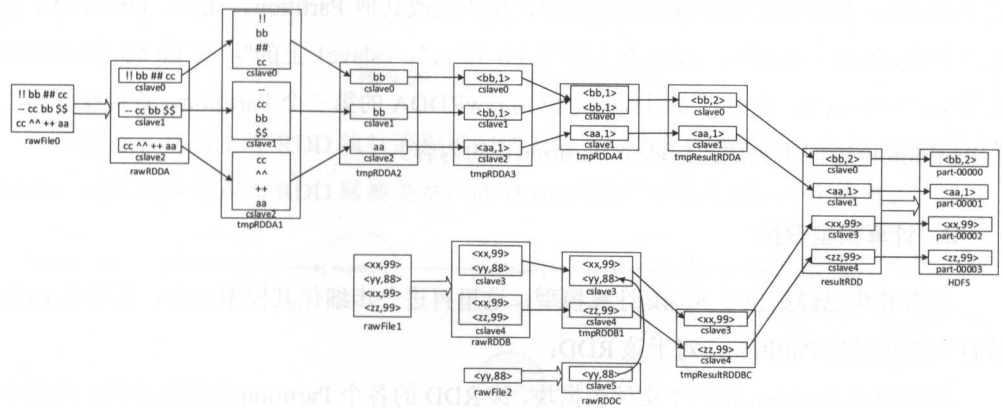


图 9-28 执行过程

显然，容易看出，对于 Spark 来说：

⑦当 Map 范式作用于 RDD 时，不会改变前后两个 RDD 内 Partition 数量；当 partitionBy、union 作用于 RDD 时，会改变前后两个 RDD 内 Partition 数量。

图中的 Map 范式：tmpRDDA2→tmpRDDA3 等都可以例证本结论；图中 tmpRDDA3→tmpRDDA4、tmpResultRDDA.union(tmpResultRDDBC)操作后，前后 Partition 数量发生改变。

⑧将 RDD 持久化到 HDFS 时，RDD 对应一个文件夹，隶属于本 RDD 的每个 Partition 对应于一个独立文件。

⑨各个 RDD 之间的数据，不论是 Map 范式还是 Shuffle 范式，中间结果不存入本地磁盘或 HDFS。

⑩各个 RDD 之间组成一个有向无环图，这个 DAG 图可以有多个（本例中为三个）输入源。

⑪可以将连续 RDD 转换操作写成一个“点”系列操作。

在“场景 1”中，遇到两个 RDD 转换时，编者是分开写的，不过本场景中，编者则是用“.”将一系列转换连到一起，即代码中的：

```
var tmpResultRDDA=rawRDDA.flatMap(xx).filter(xx).map(xx).partitionBy(xx).groupByKey().map(xx)
```

事实上, Spark 推荐这种写法, 而且中间最好不在申明新的变量, 比如:

```
rawRDDA.flatMap(line=>line.split(" "))
```

可直接写成:

```
rawRDDA.flatMap(_>.split(" "))
```

这样的代码执行起来会更加高效。

⑫可以通过 RDD 实现数据细颗粒操作。

虽然本例中并未体现该特征, 不过, 从代码层次来说, 完全可以定位到 RDD 的某特定 Partition, 直接对此 Partition 进行操作, 而不更改其他 Partition。比如, rawRDDA 拥有三个 Partition (分别为: cslave0 上的"!! bb ## cc"、cslave1 上的"-- cc bb \$\$"和 cslave2 上的"cc ^^ ++ aa"), 完全可以用代码定位到 rawRDDA 的第三个 Partition, 直接操作第三个 Partition 内容而不去更改前两个 Partition (的内容)。

2. 计算模型理论

本节开头已经给出了 Spark 计算模型, 这里再进一步细化其模型内容, 假设某 RDD 持有一定数量的 Partition, 对于该 RDD:

①物理上 Partition 是一个实体数据块, 该 RDD 的各个 Partition 会默认分散到集群中各台机器上。

②如果需要对此 RDD 进行若干次本地计算那就进行若干次本地计算 (Map), 计算时, 其并行化点是, 各机器 (实际上是 Executor) 处理存储于本机的 Partition。

③如果需要对该 RDD 内各 Partition 进行若干次重组 (如将 Partition2 的部分数据分给 Partition7), 那就进行若干次 (跨网络) 重组 (Shuffle)。

④对于 Shuffle 后 RDD, 如果还需要进行若干次 Map 或 Shuffle 操作, 那就进行若干次 Map 或 Shuffle 操作。

⑤可按业务需求, 对上述②、③、④中的 RDD 进行任意组合, 直接任务结束, 不过组合后的 RDD 必须是一个有向无环图。

⑥对于这个由 RDD 构成的有向无环图, 其可以有多个输入源。

以“场景 3”为例, 显然该场景中不仅有多个输入源 (rawRDDA、rawRddb、rawRDDC), 有典型的 Map 运算 (如 tmpRDDA→tmpRDDA1), 有 Partition 数量发生改变的 Shuffle 运算 (tmpRDDA3→tmpRDDA4), 还有将 Partition 分散于各机的并行化运算 (sc.parallelize("数据",3)), 有对两个 RDD 进行组合的 subtract、union 运算, 除了诸如将三个 RDD 进行同时组合的 RDD.cogroup(otherRDD1, otherRDD1)、对 RDD 某特定 Partition 进行操作的运算外, 本场景可以说应有尽有, 下面再次给出实例执行流程图 (图 9-29), 请读者务必仔细理解, 掌握本质。

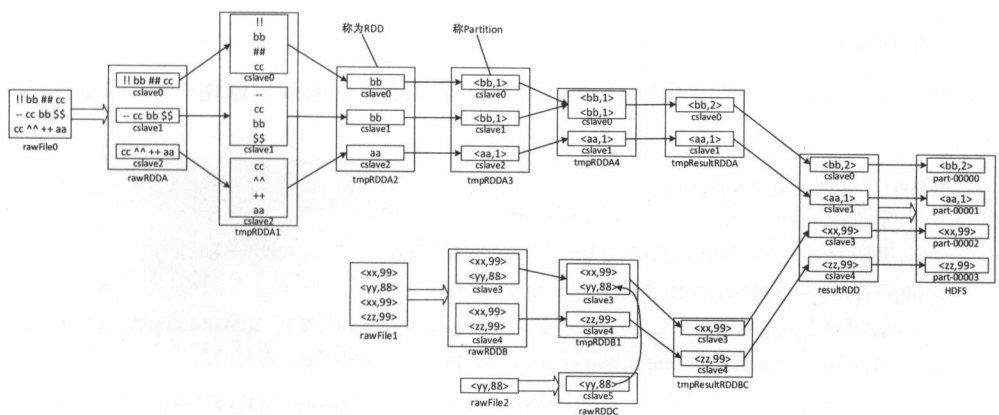


图 9-29 Spark 计算模型特例

实际上，可将上述 RDD 执行流程可归纳为如下有向无环图（图 9-30），图中顶点为 RDD，前后两个顶点为 RDD 转换关系，此 DAG 共有三个输入源，一个终点。

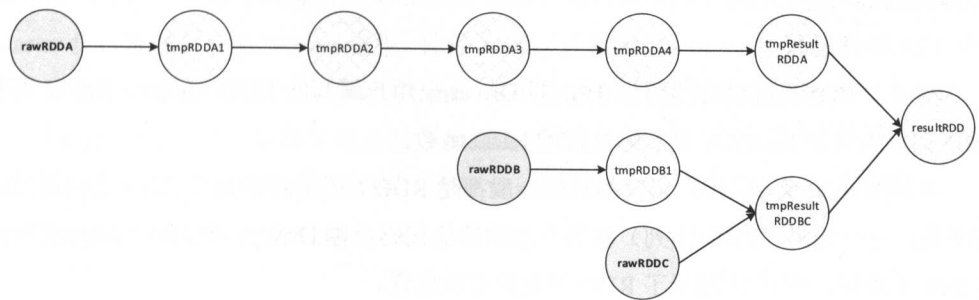


图 9-30 RDD 有向无环图

从图 9-30 中可以看出，Spark 的计算模型是：依托 RDD 作为（逻辑上）分布式存储中心和并行处理中心，以有向无环图方式来执行用户编写的 RDD 序列。

3. RDD

正如 Spark 计算模型中所述，RDD 是 Spark 核心。向下 RDD 以一组 Partition 方式将数据分散到集群中，向上 RDD 提供了一系列用户级别操作接口，RDD 的 Partition 更是并行化所在地。不过在 Spark 程序执行过程中，好像看不到 RDD 存在的痕迹，这是因为 RDD 是数据层、用户 API 层、并行执行层，这三者实际结合体，必须高度抽象。用户编写的 RDD 序列，实际上就是 Driver 进程里，用户程序的执行流程。

下面编者分别从逻辑层面和物理层面上，讲述 RDD 所在位置；请以图例方式给出 RDD 在 Spark 框架，Spark-App 里所处位置。

(1) 逻辑层面

逻辑上, Driver 中包含 SparkContext 实例, 而 SparkContext 里则由一系列 RDD 序列和其他上下文对象组成, 以下述代码为例:

```
val conf = new SparkConf()
val sc: SparkContext = new SparkContext(conf)
val rawRDDA = sc.parallelize(List("!! bb ## cc", "% % cc bb % %", "cc && ++ aa"), 3)
val tmpRDDA1 = rawRDDA.flatMap(line => line.split(" "))
val tmpRDDA2 = tmpRDDA1.filter(allWord => {allWord.contains("aa") || allWord.contains("bb")})
val tmpRDDA3 = tmpRDDA2.map(word => (word, 1))
```

程序中的 RDD 转换序列即: rawRDDA→tmpRDDA1→tmpRDDA2→tmpRDDA3。由 Driver 控制并向各个 Executor 分发; 程序中的某具体动作, 如 rawRDDA.flatMap()、tmpRDDA1.filter()、tmpRDDA2.map(), 这些动作由 Executor 执行, 故 RDD 逻辑位置是: Driver 里用户编写的业务程序代码、Driver 里各 RDD 调度模块和 Executor 里实际执行的 RDD 函数。

(2) 物理层面

物理上 RDD 位置和逻辑上一样, 即 Driver 里用户编写的 RDD Graph、Driver 里各 RDD 调度模块和 Executor 里实际执行的 RDD 函数。

正如图 9-30 所示, 用户编写的代码一般都是 RDD 序列, 前后两个 RDD 之间通过函数转换; 用户代码 (RDD 序列) 和客户端的其他代码合称 Driver, 即对应的物理实体就是 Driver 进程, 图 9-31 演示了 RDD 所处的层次位置。

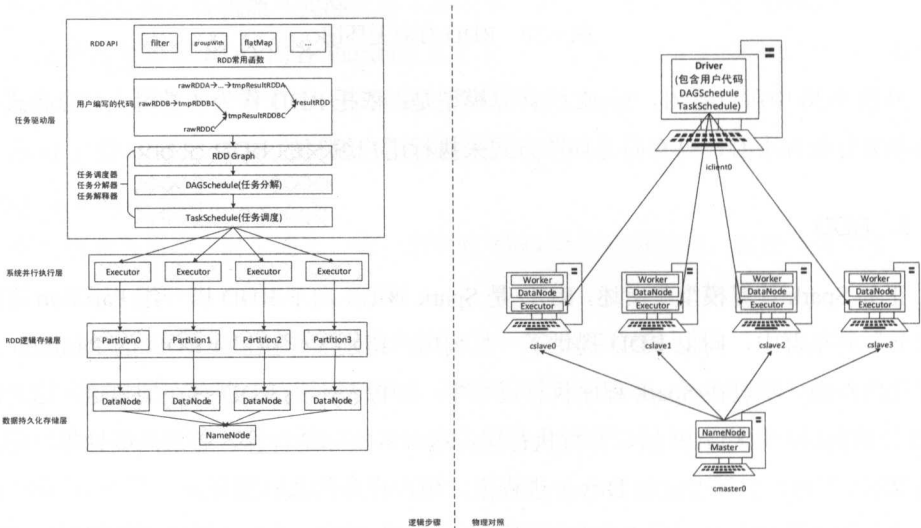


图 9-31 RDD 所处位置示意图

任务提交后，Driver 进程里的 RDD Graph 模块、DAGSchedule 模块会先后将用户编写的 RDD 序列拆分成 Executor 能够执行的逻辑单元 StageDAG-RDD。最后 TaskSchedule 将各个 StageDAG-RDD 依次发往 Executor 上执行。

9.1.5 工作机制

Spark 的工作机制是启动集群资源管理器、用户编写 Spark-App、用户提交 Spark-App、集群并行计算用户提交的 Spark-App，这一系列流程。

1. 集群资源管理层

集群资源管理器（以 Standalone 为例）采用 master/slave 架构，主进程为 Master，从进程为 Worker。集群中只能有一个 Master 进程、每台从属机上都要部署 Worker 进程。Master 负责管理所有的 Worker 进程，此外 Master 还负责 Client 的任务注册。启动步骤如下，注意在启动过程中，无须启动 Client，其不是驻守进程，和集群资源管理器无关：

Step1 主节点启动资源管理主服务 Master 进程。

Step2 各 slave 机启动资源管理从进程 Worker 进程。

Step3 各 Worker 会自动到 Master 注册自己。

2. 用户编写 Spark-App

用户再编写 Spark-App 时实质上就是将业务流编写成 RDD 序列，Driver 进程会将这个 RDD 序列分配到各机上执行。

当客户端向 Standalone 提交 Spark-App 时，实质上是客户端的 Driver 进程向 Master 进程注册本应用程序，Master 判断任务合法后，会要求各 Worker 启动相应 Executor 来执行 Spark-App。

3. Spark-App 应用程序执行

Spark-App 的执行进程依旧是 master/slave 架构，主进程为 Driver，从进程为 Executor，Driver 会将用户编写的 RDD 序列里的具体操作发往各个 Executor，这些 Executor 则是任务的执行实体。

在 Spark 集群上执行代码的过程就是启动 Driver 进程和隶属于本 Driver 的一系列 Executor，在 Spark-App 实际执行过程中，任务流调度由 Driver 控制，某具体任务的实际执行由各个 Executor 执行，其基本流程如下：

Step1 用户编写 Spark-App 代码, 须将用户编写的代码和 Client 包打包到一起。

Step2 Client 向 Master 提交任务。

Step3 Master 在确认 Spark-App 合法后, 要求各 Worker 启动相应 Executor。

Step4 各个 Executor 向 Driver 注册自己。

Step5 Driver 指挥各 Executor 并行处理本次 Spark-App。

4. 完整执行流

用户编写好一系列 RDD 串后, 这些 RDD 串会和 Client 包打包到一起。也就是在 Client 提交 RDD 串之前, Client 里至少包含 Spark-Client 协议实现体、用户编写的 Spark 代码、任务调度等各个模块。当 Client 向 Master 提交任务时, Client 须使用 Spark-Client 协议向 Master 提交任务, 使用任务调度模块将用户代码在各个 Executor 上调度执行, 图 9-31 即为 Standalone 模式下 Spark-App 任务执行流程图, 从图中能够清晰看出用户编写的 RDD 序列在 Driver 进程中, Driver 会将用户代码调度到各 Executor 上执行。在图 9-31 上执行 Spark-App 时, 其执行过程可描述如下:

Step1 用户调用 RDD API 接口, 编写类似“场景 3”中程序代码。

Step2 用户调用 Spark 提交任务接口, 向 Master 提交任务。

Step3 Master 接收客户端提交的 Spark 任务, 接着 Master 要求各 Worker 启动 Executor。

Step4 用户编写的代码和提交任务的客户端统一称 Driver, 各 Executor 向 Driver 注册。

Step5 RDD Graph 将用户的 RDD 串组织成 DAG-RDD。

Step6 DAGSchedule 以 Shuffle 为原则 (即遇 Shuffle 就拆分) 将 DAG-RDD 拆分成一系列 StageDAG-RDD (StageDAG-RDD0→StageDAG-RDD1→StageDAG-RDD2→...)。

Step7 RDD 将通过访问 NameNode, 将 DataNode 上的数据块装入 RDD 的 Partition。

Step8 TaskSchedule 将 StageDAG-RDD0 发往隶属于本 RDD 的所有 Partition 执行, 在 Partition 执行过程中, Partition 所在机上的 Executor 优先执行本 Partition。

Step9 TaskSchedule 将 StageDAG-RDD1 发往隶属于本 RDD (已改变) 的所有 Partition 执行。

Step10 重复上述步骤, 直至执行完所有 Stage-DAG-RDD。

9.1.6 其他特性

1. 以应用程序来隔离资源

每个应用程序都有自己一系列 Executor 进程, 站在单个 Spark-App 的角度上, 这些

Executor 会协作完成该任务；站在某特定 Executor 的角度上，该 Executor 会以多线程复用方式运行该 Spark-App 分配来的所有 Task。这点和 MapReduce 不相，比如如某个由 Map→Reduce→Reduce 构成的 ML-App，有十个 Slave 同时执行该任务，当用 MapReduce 框架执行时，slave3 机（其他机器相同）须启动 Map 进程、Reduce 进程、Map 进程、Reduce 进程，四个进程顺序执行该任务，而 Spark 则使用一个 Executor 进程完成这四个操作。

通过采用“每个应用程序都有自己一系列 Executor 进程”这种机制确保了从调度层（各个不同的 Driver）和执行层（附属于该 Driver 的一系列 Executor）隔离应用程序。不过这也导致了当不借助第三方存储系统时，不同的 Spark-App 很难实现数据共享。

2. 集群管理器

不论是 Standalone 还是 Third-Platform 模式，Spark-App 本身都不会感知到集群存在。唯一需要保证的就是确保 Driver 进程和附属于该 Driver 的一系列 Executor 进程能够相互通信，即可完成 Spark-App。

比如当采用 Standalone 模式时，Master 和 Worker 分别负责管理集群资源和本机资源，不会参与 Spark-App 任何计算，也就是 Spark-App 还是必须由 Driver 和 Executor 协作完成；当采用 Third-Platform 模式时，以 YARN 为例，ResourceManager 和 NodeManager 分别负责管理集群资源和本机资源，不会参与 Spark-App 任何计算，也就是 Spark-App 还是必须由 SparkAPPMaster 和 SparkExecutor 协作完成（此时集群已不存在 Master 和 Worker 进程）。

3. 网络可达

在 Spark-App 的整个生命周期内，必须确保 Driver 进程时刻侦听各 Executor 的远程连接，这是因为 Driver 进程负责控制 Spark-App 整个程序流，故须确保 Driver 机和 Executor 属于同一个 Network。

此时，Driver 机最好离 Executor 机足够近，比如将 Driver 机和 Executor 机运行于同一个本地机房。如果不得不以远程方式运行 Driver，最好将 Driver 运行在 Worker 里。

9.2 Spark 接口

Spark 接口指的是用户取得 Spark 服务的途径，下面先讲述常见的 Spark 接口，接着直接讲述 Web 接口。

1. 接口汇总

作为大数据处理领域最流行的并行计算框架,针对不同的上层应用,Spark 框架提供了七类统一访问接口,分别为:

- Spark 自带 Web 接口
- Spark Shell 接口
- Spark API 接口
- Spark SQL 接口
- Spark Streaming 接口
- Spark MLlib 接口
- Spark Graphx 接口

Web 接口主要为管理员提供,从该页面上,管理员能看到正在执行的和已完成的所有 Spark-App 执行过程中的统计信息,该页面只支持读、不支持写操作。

Shell 接口主要针对 Spark 程序员和 Spark 数据分析师,通过 Shell 接口,程序员能够向 Spark 集群提交 Spark-App、查看正在运行的 Spark-App;数据分析师可以通过 Shell 接口以交互式方式对数据进行实时分析,9.3 节重点 Spark 的 Shell 接口。

Spark API 面向 Java、Scala、Python 和 R 工程师、分析师,程序员可以通过该接口编写 Spark-App 用户层代码 MRApplicationBusinessLogic,9.4 节将讲述 Spark 编程。

至于 SQL、Streaming、MLlib 和 Graphx 接口,9.5 节会依次给出其示例,但不会深入讲解。

2. 实战 Spark Web

由于 Spark Web 接口内容较少,此处直接讲解。当只开启资源管理器时,集群中只显示资源管理器主界面;当用户向集群提交 Spark-App 后,Driver 进程也会开启一个 Web 界面。

1) 集群资源管理层

Spark 集群资源管理器的默认 Web 地址是“MasterIP:8080”,在 littleCstor 中 Master 服务部署在 cmaster0 上,故在浏览器中地址栏输入“cmaster0.cloudlab.njupt.edu:8080”即可进入 Spark 资源管理器主界面(图 9-32)。

刚启动的集群中没有正在执行的 Spark-App,也没有运行完成的 Spark-App[图 9-32(a)];当用户向 Spark 集群提交任务后,主界面上会显示正在运行的 Spark-App。

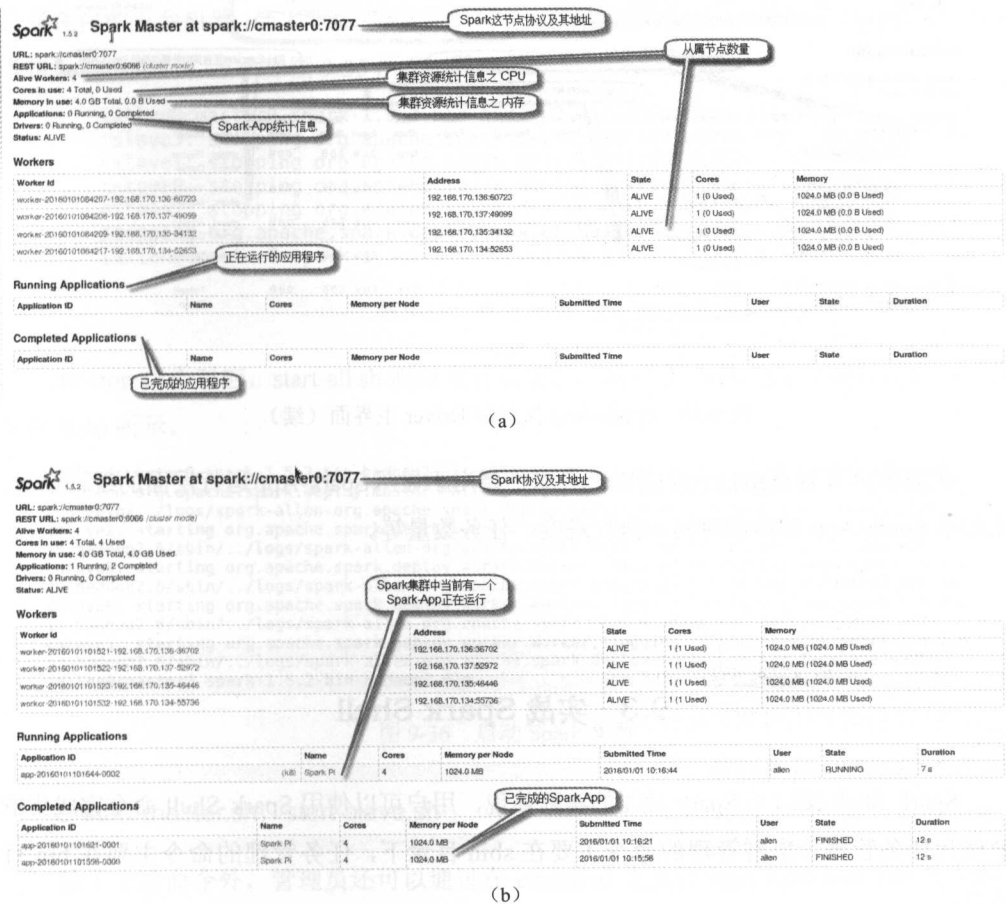


图 9-32 Spark Web 主界面

2) 应用程序执行层

当客户端向 Spark 集群提交 Spark-App 后，客户进程 Driver 也会启动一 Web 页面 (图 9-33)，该界面地址为“DriverIP:4040”，由于编者使用 iclient0 向 Spark 集群提交任务，故该任务地址是“iclient0:4040”。

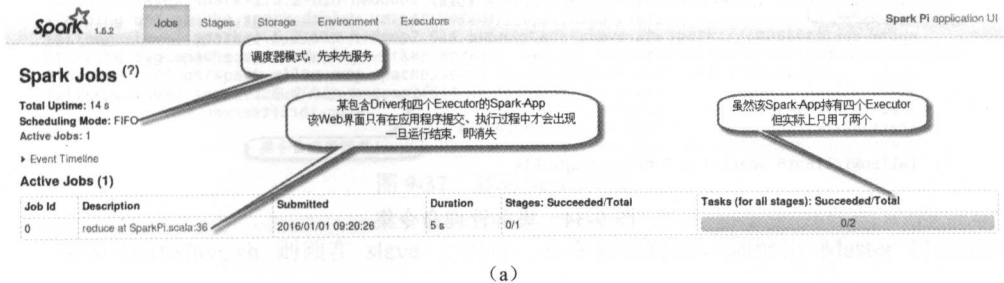


图 9-33 Spark-App 执行层 Driver 主界面

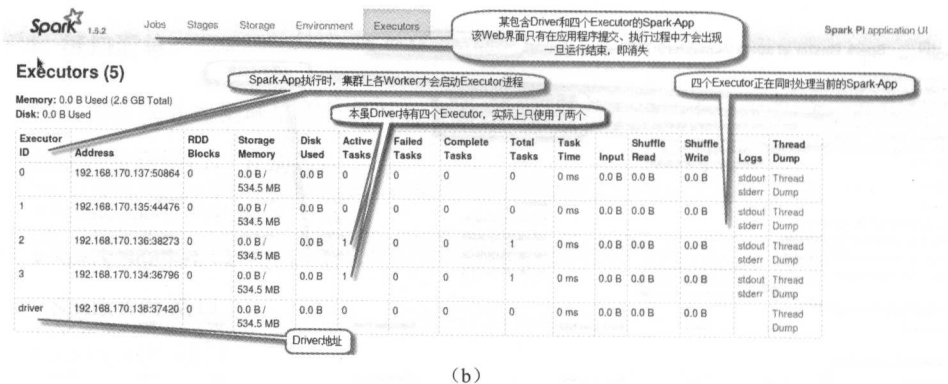


图 9-33 Spark-App 执行层 Driver 主界面 (续)

从图9-33可以看出Driver的Web界面主要用于显示Spark-App运行层面的统计信息，比如本Spark-App的启动时间、执行进度、任务数量等。

9.3 实战 Spark Shell

Spark Shell 接口是 Spark 功能的实际体现，用户可以使用 Spark Shell 命令完成集群管理和任务管理。集群管理的命令主要在 sbin 目录下，任务管理的命令主要在 bin 目录下。

9.3.1 集群管理

可以通过“sbin”目录下的命令启动或关闭集群中的某服务或者是整个集群，图 9-34 为“sbin”下的所有命令集合。

```
[allen@iclient0 spark-1.5.2-bin-hadoop2.6]$ ls sbin/
slaves.sh
spark-config.sh
spark-daemon.sh
spark-daemons.sh
start-all.sh
start-history-server.sh
start-master.sh
start-mesos-dispatcher.sh
start-mesos-shuffle-service.sh
start-shuffle-service.sh
start-slave.sh
start-slaves.sh
start-thriftserver.sh
stop-all.sh
stop-history-server.sh
stop-master.sh
stop-mesos-dispatcher.sh
stop-mesos-shuffle-service.sh
stop-shuffle-service.sh
stop-slave.sh
stop-slaves.sh
stop-thriftserver.sh
```

图 9-34 集群管理命令集

1. stop-all.sh, start-all.sh

该命令主要用来关闭整个 Spark 集群,注意命令执行前提是 cmaster0 到包括 cmaster0

自身在内的五台机器 ssh 时皆无需密钥。编者已经在 littleCstor 上开启了 Spark，故当编者执行该命令时，显示如图 9-35 所示。

```
[allen@cmaster0 spark-1.5.2-bin-hadoop2.6]$ sbin/stop-all.sh
cslave3: stopping org.apache.spark.deploy.worker.Worker
cslave1: stopping org.apache.spark.deploy.worker.Worker
cslave0: stopping org.apache.spark.deploy.worker.Worker
cslave2: stopping org.apache.spark.deploy.worker.Worker
stopping org.apache.spark.deploy.master.Master
[allen@cmaster0 spark-1.5.2-bin-hadoop2.6]$
```

关闭 Spark 集群
★★ 只能在 cmaster0 上执行
一个 Master 机
四个 Worker 机

图 9-35 关闭 Spark 集群

和 stop 命令对应，start-all.sh 则用来开启集群，编者在 littleCstor 上执行该命令过程如图 9-36 所示。

```
[allen@cmaster0 spark-1.5.2-bin-hadoop2.6]$ sbin/start-all.sh
starting org.apache.spark.deploy.master.Master, logging to /home/allen/spark-1.5.2-bin-hadoop
2.6/sbin/../logs/spark-allen-org.apache.spark.deploy.master.Master-1-cmaster0.out
cslave2: starting org.apache.spark.deploy.worker.Worker, logging to /home/allen/spark-1.5.2-b
in-hadoop2.6/sbin/../logs/spark-allen-org.apache.spark.deploy.worker.Worker-1-cslave2.out
cslave3: starting org.apache.spark.deploy.worker.Worker, logging to /home/allen/spark-1.5.2-b
in-hadoop2.6/sbin/../logs/spark-allen-org.apache.spark.deploy.worker.Worker-1-cslave3.out
cslave0: starting org.apache.spark.deploy.worker.Worker, logging to /home/allen/spark-1.5.2-b
in-hadoop2.6/sbin/../logs/spark-allen-org.apache.spark.deploy.worker.Worker-1-cslave0.out
cslave1: starting org.apache.spark.deploy.worker.Worker, logging to /home/allen/spark-1.5.2-b
in-hadoop2.6/sbin/../logs/spark-allen-org.apache.spark.deploy.worker.Worker-1-cslave1.out
[allen@cmaster0 spark-1.5.2-bin-hadoop2.6]$
```

启动 spark 集群，★★ 只能在 cmaster0 上执行
启动 Master
启动四个 Worker

图 9-36 启动 Spark 集群

2. start-master.sh, start-slave.sh

除了上述命令外，管理员还可以通过在 cmaster0 上执行 start-master.sh 和所有 slave 上执行 start-slave.sh 组合完成集群启动，图 9-37(a)演示了在 cmaster0 上使用 start-master.sh 和 stop-master.sh 命令，前一个命令用于启动 Master 服务，后一个则用于关闭 Master 服务。

```
[allen@cmaster0 spark-1.5.2-bin-hadoop2.6]$ sbin/stop-master.sh
stopping org.apache.spark.deploy.master.Master
[allen@cmaster0 spark-1.5.2-bin-hadoop2.6]$ sbin/start-master.sh
starting org.apache.spark.deploy.master.Master, logging to /home/allen/spark-1.5.2-bin-hadoop
2.6/sbin/../logs/spark-allen-org.apache.spark.deploy.master.Master-1-cmaster0.out
[allen@cmaster0 spark-1.5.2-bin-hadoop2.6]$
```

cmaster0 上执行
关闭 Master 服务
cmaster0 上执行，启动 Master 服务

(a)

```
[allen@cslave2 spark-1.5.2-bin-hadoop2.6]$ sbin/stop-slave.sh spark://cmaster0:7077
stopping org.apache.spark.deploy.worker.Worker
[allen@cslave2 spark-1.5.2-bin-hadoop2.6]$ sbin/start-slave.sh spark://cmaster0:7077
starting org.apache.spark.deploy.worker.Worker, logging to /home/allen/spark-1.5.2-bin-hadoop
2.6/sbin/../logs/spark-allen-org.apache.spark.deploy.worker.Worker-1-cslave2.out
[allen@cslave2 spark-1.5.2-bin-hadoop2.6]$
```

cslave2: 关闭 Worker
cslave2: 开启 Worker

(b)

图 9-37 启动 Spark 集群

命令 start-slave.sh 则须在 slave 上执行，且在其执行时，须给出 Master 协议地址“spark://cmaster0:7077”。“sbin”下的其他 Shell 命令和本命令类似，请读者自行练习。

9.3.2 任务管理

可以通过“bin”目录下的命令向 Spark 集群提交 Spark-App，管理 Spark-App。图 9-38 罗列了“bin”目录下所有命令（cmd 后缀为 Windows 平台脚本）。

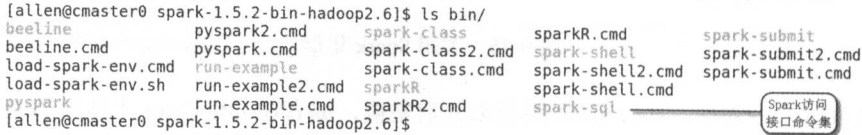


图 9-38 启动 Spark 集群

1. spark-submit

该命令用来向 Spark 集群提交 Spark-App，第一个命令用于显示 spark-submit 使用方式，由于输出太多，编者并未截图。

```
[allen@iclient0 spark-1.5.2-bin-Hadoop2.6]$ bin/spark-submit --help
[allen@iclient0 spark-1.5.2-bin-Hadoop2.6]$ bin/spark-submit --master spark://cmaster0:7077
--class org.apache.spark.examples.SparkPi lib/spark-examples-1.5.2-Hadoop2.6.0.jar
[allen@iclient0 spark-1.5.2-bin-Hadoop2.6]$ bin/spark-submit --class org.apache.spark.examples.SparkPi
--master yarn-cluster --num-executors 3 --driver-memory 1g --executor-memory 1g
--executor-cores 1 lib/spark-examples*.jar 10
```

第二个命令则是使用 spark-submit 命令向 Spark 集群提交一个 Spark-App (PI)，注意在书写命令时必须要有参数“--master spark://cmaster0:7077”，若不写明该参数，spark-submit 会启动本地模式（而不是集群模式）来计算该任务。

同理，第三个命令则是使用 spark-submit 向 YARN 集群提交 Spark-App (PI)，和第二个命令相同，书写本命令时必须加上参数“--master yarn-cluster”，读者可能会奇怪，为何参数中没有写明 YARN 集群协议地址，这是因为 Spark 的配置文件中已经指向了 Hadoop 集群，该命令会自动读取，故无须再配置。本命令执行的前提是 HDFS 和 YARN 集群都已开启。

2. pyspark

该命令用于以交互式方式编写并执行 Spark-App，且书写语法为 Python，下面的命令用于进入交互式执行器，进入执行器后，数据分析师可编写 Python 语句实时操作 Spark。

```
[allen@iclient0 spark-1.5.2-bin-hadoop2.6]$ bin/pyspark --master spark://cmaster0:7077
```

注意书写上述命令时，必须写明“--master spark://cmaster0:7077”，否则 pyspark 会默认进入单机模式，而不会进入集群模式。

既然能以 Python 方式交互式编写 Spark-App, OS 上肯定有 Python 编译器, 当前 Spark-App 指定的 Python 为 Python2.6。事实上, OS 上必须有 Python, 否则连 Ambari 都不可能成功安装。

3. sparkR

该命令用于以交互式方式编写并执行 Spark-App, 且书写语法为 R, 下面的命令用于进入交互式执行器, 进入执行器后, 数据分析师可编写 R 语句实时操作 Spark。

```
[allen@iclient0 spark-1.5.2-bin-hadoop2.6]$ bin/sparkR --master spark://cmaster0:7077
```

注意书写上述命令时, 必须写明 “--master spark://cmaster0:7077”, 否则 sparkR 会默认进入单机模式, 而不会进入集群模式。

和 Python 不同, OS 上一般不会安装 R, 故用户须安装 R 环境, 才可使用 “sparkR”。

4. spark-shell

该命令用于以交互式方式编写并执行 Spark-App, 且书写语法为 Scala, 下面的命令用于进入交互式执行器, 进入执行器后, 数据分析师可用 Scala 语句以交互式方式编写并执行 Spark-App:

```
[allen@iclient0 spark-1.5.2-bin-hadoop2.6]$ bin/spark-shell --master spark://cmaster0:7077
```

注意书写上述命令时, 必须写明 “--master spark://cmaster0:7077”, 否则 spark-shell 会默认进入单机模式, 而不会进入集群模式。

和由于 Spark 就是使用 Scala 开发的, 而 Scala 的实际上就是在 JVM 中执行的, 故无须安装 Scala。

5. run-example

该脚本用于运行 Spark 示例程序, 实际上, 该脚本内部调用了 spark-submit, 读者不必掌握该命令。

6. spark-class

这是底层的服务提交方式, spark-submit 命令内部调用了该脚本, 一般不常用。

7. spark-sql

该命令用于以交互式方式编写并执行 Spark SQL, 且书写语法类 SQL, 下面的命令用于进入交互式执行器, 进入执行器后, 数据分析师可用类 SQL 语句以交互式方式编写并执行 Spark-App:

```
[allen@iclient0 spark-1.5.2-bin-hadoop2.6]$ bin/spark-sql --master spark://cmaster0:7077
```

注意书写上述命令时, 必须写明 “--master spark://cmaster0:7077”, 否则 spark-shell 会默认进入单机模式, 而不会进入集群模式。

由于默认安装的 Spark 已经包含了 Spark SQL，故无须再安装其他组件，直接执行即可。

9.4 实战 Spark 编程之 RDD

在 9.1.1 节编者曾指出“Spark 通过 RDD 将框架功能和操作函数优雅地结合起来，大大方便了用户编程”，没有编写过 Spark 代码的用户一定感觉 RDD “深不可测”，实际上，它非常简单，只需一个 Spark 代码，你将深刻体会到原来 RDD 如此“可爱”。

不管是“深不可测”还是“可爱”，可以说 RDD 就是 Spark 核心，是编写 Spark-App 永远避不开的核心内容，请读者务必掌握。

本章所有代码都在交互式执行器下执行，在 iclient0 上使用 spark-shell 命令进入 Spark 交互式执行器，注意务必加上“--master spark://cmaster0:7077”选项，否则只进入默认的本地模式而不是集群模式。

```
[allen@iclient0 spark-1.5.2-bin-hadoop2.6]$ bin/spark-shell --master spark://cmaster0:7077
```

交互式执行器界面如图 9-39 所示，由于输出太多，此处并未给出中间过程：

```
allen@iclient0 spark-1.5.2-bin-hadoop2.6] version 1.5.2

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_20)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.
SQL context available as sqlContext.

scala>
```

图 9-39 进入交互式执行器

9.4.1 RDD 属性

RDD (Resilient Distributed Datasets)，站在程序员角度上，可将 RDD 看成一系列 Partitions 的集合。站在 Spark 框架角度上，RDD 则是一个个需要并行执行的操作。

一个 RDD 持有一系列 Partition，这些 Partition 则均匀分散于各 slave 节点上，用于存储实际数据。下面的程序完成在内存中构建“1~9”9 个数据并设置其并行化因子为 3，最后使用自定义函数查看 PartitionID 及其存储在该 Partition 上的数据。

```
// 在内存中构建“1~9”9 个数据，将其分散到 3 个 Partition 上
val z = sc.parallelize(List(1,2,3,4,5,6,7,8,9),3)
//自定义查看输入分区函数
```



```
def myfunc(index: Int, iter: Iterator[(Int)]) : Iterator[String] = {
  iter.toList.map(x => "[partID:" + index + ", val: " + x + "]").iterator
}
//使用该函数查看 Partition 及其 ID
z.mapPartitionsWithIndex(myfunc).collect.foreach(println)
// 查看本 RDD 持有的 Partition 数量
z.partitions.length
//存储输入数据至 HDFS
z.mapPartitionsWithIndex(myfunc).saveAsTextFile("test/hh")
```

程序中 saveAsTextFile()一句执行的前提是集群指向了 HDFS（且 HDFS 已经开启），z.partitions.length 一句则用于显示本 RDD 持有的 Partition 数量，整个程序执行流程如图 9-40（a）所示。

类似上述代码，我们也可构建字符串序列，下面的代码即完成上述构建字符串功能，注意由于字符串和 Int 不同，故自定义函数时，须将 Int 改成 String。

```
// 在内存中构建 aa-kk 10 个数据并将这 10 个数据分散到 4 个 Partition 上
val z = sc.parallelize(List("aa","bb","cc","dd","ee","ff","gg","hh","ii","kk"),4)
//自定义查看输入分区函数
def myfunc(index: Int, iter: Iterator[(String)]) : Iterator[String] = {
  iter.toList.map(x => "[partID:" + index + ", val: " + x + "]").iterator
}
//使用该函数查看 Partition 及其 ID
z.mapPartitionsWithIndex(myfunc).collect.foreach(println)
// 查看本 RDD 持有的 Partition 数量
z.partitions.length
//存储输入数据至 HDFS
z.mapPartitionsWithIndex(myfunc).saveAsTextFile("test/hh-1")
```

程序中 save.AsTextFile()一句执行的前提是集群指向了 HDFS（且 HDFS 已经开启），z.partitions.length 一句则用于显示本 RDD 持有的 Partition 数量，整个程序执行流如图 9-40（b）所示。

```
scala>
scala> val z = sc.parallelize(List(1,2,3,4,5,6,7,8,9),3)
z: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:21
scala> def myfunc(index: Int, iter: Iterator[(Int)]) : Iterator[String] = {
  | iter.toList.map(x => "[partID:" + index + ", val: " + x + "]").iterator
  | }
myfunc: (index: Int, iter: Iterator[Int])Iterator[String]
scala> z.mapPartitionsWithIndex(myfunc).collect.foreach(println)
[partID:0, val: 1]
[partID:0, val: 2]
[partID:0, val: 3]
[partID:1, val: 4]
[partID:1, val: 5]
[partID:1, val: 6]
[partID:2, val: 7]
[partID:2, val: 8]
[partID:2, val: 9]
scala> z.partitions.length
res1: Int = 3
scala>
```

从内中构建9个数据，申请并行化因子为3

自定义函数，用于深入查看Partition索引及其内容

使用刚才自定义的函数，查看该Partition索引及其内容

PartID: 0 存储了 1,2,3 三个数据

PartID: 1 存储了 4,5,6 三个数据

PartID: 2 存储了 7,8,9 三个数据

查看本RDD持有的Partition数量

本RDD持有 3 个Partition

(a)

```

scala>
scala> val z = sc.parallelize(List("aa","bb","cc","dd","ee","ff","gg","hh","ii","kk"),4)
z: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[0] at parallelize at <console>:21

scala> def myfunc(index: Int, iter: Iterator[(String)]) : Iterator[String] = {
  |   iter.toList.map(x => "[partID:" + index + ", val: " + x + "]").iterator
  | }
myfunc: (index: Int, iter: Iterator[String])Iterator[String]

scala> z.mapPartitionsWithIndex(myfunc).collect.foreach(println)
[partID:0, val: aa]
[partID:0, val: bb]
[partID:1, val: cc]
[partID:1, val: dd]
[partID:1, val: ee]
[partID:2, val: ff]
[partID:2, val: gg]
[partID:3, val: hh]
[partID:3, val: ii]
[partID:3, val: kk]

scala> z.partitions.length
res1: Int = 4

scala>

```

从内存中构建10个数据，申请并行化因子为4

自定义函数，用于深入查看Partition索引及其内容，注意函数签名已由Int改成String

使用刚才自定义的函数，查看该Partition索引及其内容

PartID: 0 存储了 aa,bb 两个数据

PartID: 1 存储了 cc,dd,ee 三个数据

PartID: 2 存储了 ff,gg 两个数据

PartID: 3 存储了 hh,ii,kk 三个数据

查看本RDD持有的Partition数量

本RDD持有 4 个Partition

(b)

图 9-40 RDD 所持有的 Partition

实际上，此处的“z”就是一个 RDD，RDD 有两种构建方式。

(1) 从持久化数据源（如 HDFS）构建 RDD

```

val conf = new SparkConf()
val sc: SparkContext = new SparkContext(conf)
val z = sc.textFile("/user/allen/in/note3.txt",4)

```

(2) 从内存数据源构建 RDD

```

val conf = new SparkConf()
val sc: SparkContext = new SparkContext(conf)
val z = sc.parallelize(List("aa","bb","cc","dd","ee","ff","gg","hh","ii","kk"),4)

```

图 9-40 中之所以没有前两句是因为在 spark-shell 命令启动过程中，启动命令内部已经执行了这两句：

```

val conf = new SparkConf()
val sc: SparkContext = new SparkContext(conf)

```

9.4.2 并行化证明 RDD、调试 RDD

下面以 RDD 上一个非常复杂的函数 `aggregate` 为例，讲述调试 RDD，在调试过程中，读者将看到 RDD 的各个 Partition 是被并行处理的，若能彻底理解 RDD 调试、并行化和 `aggregate`，对 Spark 的理解将更加深入。

```

aggregate[U](zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U) => U)(implicit arg0: ClassTag[U]): U

```

函数 `aggregate` 入参为一个初始值和两个函数，初始值为函数初始参数，第一个函数将每个 Partition 上的数据规约成一个数据。第二个函数将第一个函数的结果（每个 Partition 的数据）再次规约成一个数据。需要特别注意的是，第二个函数规约时，各 Partition 结果组合过程未排序。

1. 示例

```
val z = sc.parallelize(List(3,2,1,6,5,4,3,9,7), 3)
def myfunc(index: Int, iter: Iterator[Int]) : Iterator[String] = {
  iter.toList.map(x => "[partID:" + index + ", val: " + x + "]").iterator
}
z.mapPartitionsWithIndex(myfunc).collect.foreach(println)
z.aggregate(4)((x1,y1)=>{println(x1+"::"+y1);math.max(x1, y1)}, (x2,y2)=>{println(x2+"::"+y2);
x2 + y2})
z.aggregate(4)(math.max(_ , _), _ + _)
```

交互式执行器中，代码及其执行过程，如图 9-41 所示。

```
scala>
scala> val z = sc.parallelize(List(3,2,1,6,5,4,3,9,7), 3)
z: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at parallelize at <console>:21

scala> def myfunc(index: Int, iter: Iterator[Int]) : Iterator[String] = {
  |   iter.toList.map(x => "[partID:" + index + ", val: " + x + "]").iterator
  | }
myfunc: (index: Int, iter: Iterator[Int])Iterator[String]

scala> z.mapPartitionsWithIndex(myfunc).collect.foreach(println)
[partID:0, val: 3]
[partID:0, val: 2]
[partID:0, val: 1]
[partID:1, val: 6]
[partID:1, val: 5]
[partID:1, val: 4]
[partID:2, val: 3]
[partID:2, val: 9]
[partID:2, val: 7]

scala> z.aggregate(4)((x1,y1)=>{println(x1+"::"+y1);math.max(x1, y1)}, (x2,y2)=>{println(x2+"::"+y2); x2 + y2})
res1: Int = 23
```

从内从中构建9个数据，申请并行化因子为3

自定义函数，用于查看这个函数在哪里

aggregate函数示例

aggregate函数结果

图 9-41 aggregate 函数及其结果

2. 证明并行化

在“cmaster0:8080”页面上，点击“Application ID”（本 Spark-App 的 ID 为 app-20160102122824-0007），进入本作业统计页面（图 9-42），然后依次点击各 Executor 的 stdout 链接，进入图 9-43、图 9-44、图 9-45、图 9-46。

Spark 1.5.2 Application: Spark shell

ID: app-20160102122824-0007

Name: spark shell

User: shi

Core: Unlimited (4 granted)

Executor Memory: 1024.0 MB

Submitted Date: Sat Jan 02 12:28:24 PST 2016

Status: FINISHED

Application Detail UI

Executor Summary

ExecutorID	Worker	Cores	Memory	Status	Logs
Removed Executors					
2	worker-2016010211440205-192.168.170.134-364027	1	1024	KILLED	stdout stderr
1	worker-2016010211440116-192.168.170.135-44506	1	1024	KILLED	stdout stderr
3	worker-2016010211440117-192.168.170.135-54704	1	1024	KILLED	stdout stderr
0	worker-2016010211440115-192.168.170.137-26027	1	1024	KILLED	stdout stderr

点击链接，进入图43

点击链接，进入图44

点击链接，进入图45

点击链接，进入图46

图 9-42 aggregate 函数及其结果

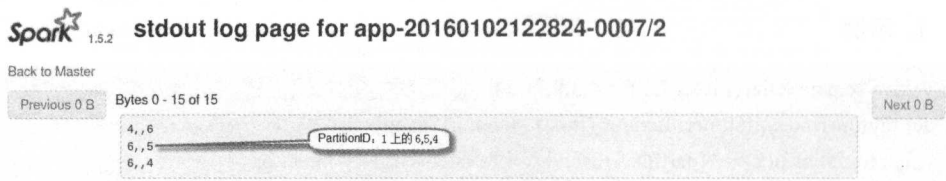


图 9-43 Executor2 的 stdout



图 9-44 Executor1 的 stdout

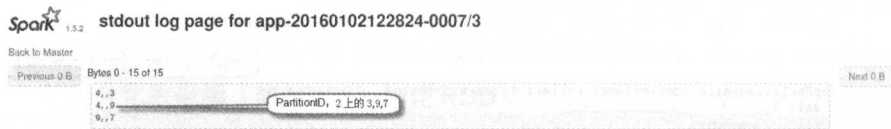


图 9-45 Executor3 的 stdout

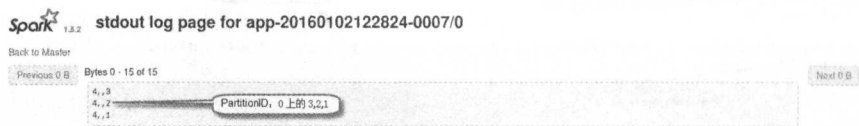


图 9-46 Executor0 的 stdout

3. 执行过程

首先, 对于第一个函数, 其处理过程如下。

基准值: 4 (图 9-41)

第一个函数: `math.max(,)` (图 9-41)

输入数据: Partition0: (3,2,1)、Partition1: (6,5,4)、Partition2: (3,9,7)三个分区 (图 9-41)

Partition0: $\max(4,3)=4, \max(4,2)=4, \max(4,1)=4$, 故 Partition0 结果为 4 (图 9-46)

Partition1: $\max(4,6)=6, \max(6,5)=6, \max(6,4)=6$, 故 Partition1 结果为 6 (图 9-43)

Partition2: $\max(4,3)=5, \max(5,9)=6, \max(6,7)=7$, 故 Partition2 结果为 9 (图 9-45)

接着, 对于第二个函数, 其处理过程如下。

基准值: 4 (图 9-41)

第二个函数: `_ + _`

输入数据: Partition0: 4, Partition1: 6, Partition2: 9

计算公式：基准值+Partition0 值+Partition1 值+Partition2 值（注，三个 Partition 值先后顺序可颠倒）

计算过程：4+4+6+9=23（注后面三个 4,6,9 先后顺序可颠倒）

4. 评价

（1）证明并行化

为了证明 Spark 在并行处理用户数据，编者特地截了 6 个图（图 9-41～图 9-46），从图 9-41 中读者应当能够清晰看到，Partition0 存储着“3,2,1”，图 9-46 则显示 Executor0 在处理 Partition0；图 9-41 中，Partition1 存储着“6,5,4”，图 9-43 则显示 Executor2 在处理 Partition1；图 9-41 中，Partition2 存储着“3,9,7”，图 9-45 则显示 Executor3 在处理 Partition2。显然，Spark 在并行处理用户数据。

由于 Spark 集群有四个从属机，而本 RDD 只有三个 Partition（图 9-41），故必有一个 Executor 未参与计算（图 9-44）。

（2）aggregate 函数

aggregate 函数的入参为一个初始值和两个函数，这两个函数的作用都是规约，其实这一过程非常复杂，请读者仔细调试如下代码，该代码会在页面上打印输出计算过程（图 9-43～图 9-46）。

```
z.aggregate(4)((x1,y1)=>{println(x1+"::"+y1);math.max(x1, y1)}, (x2,y2)=>{println(x2+"::"+y2);x2+y2})
```

实际上，在真正写 Scala 代码时，上述代码完全等同于下句：

```
z.aggregate(4)(math.max(_,_),_+_)
```

需要注意的是，Spark 的 Partition 组合时，并没有顺序，由于此处是 Int 相加，结果一致，若字符串相加，两次处理结果可能完全不同，比如下面代码：

```
val z = sc.parallelize(List(1,2,3,4,5,6), 2)
z.aggregate("b")((x1,y1)=>{ println(x1+"::"+y1);x1+y1}, (x2,y2)=>{println(x2+"::"+y2);x2 + y2})
z.aggregate("b")(_+_ ,_+_)
```

结果就可能为“bb456b123”或“bb123b456”，显然，两个 Partition 组合顺序不固定。

9.4.3 RDD 操作

RDD 上有一百个操作函数，如果全部讲解，一百页也不够，本节只选取 20 个函数，其中，各函数调试方法和证明其并行化方法和 aggregate 相同，若需要调试或证明并行化，请参考 9.4.2 节。

本节所有示例均来自 Professor Zhen He 团队在如下网址上公开的英文示例：

<http://homepage.cs.latrobe.edu.au/zhe/ZhenHeSparkRDDAPIExamples.html>

若该网址打不开,读者可在 Baidu、Bing、Google 里直接搜索:

rdd api example

一定能找到 Professor Zhen He 给出的 RDD API 示例,该网址是目前全球最有名的 RDD API 网址。在本书写作期间,编者给 Professor Zhen He 发了邮件,征得同意后,使用了如下示例,再次感谢 Professor Zhen He 团队的帮助。

1. distinct

去掉 RDD 内重复数据(前两行为代码,后一行为结果):

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2)
c.distinct.collect
res6: Array[String] = Array(Dog, Gnu, Cat, Rat)
```

2. first

查看 RDD 第一个 Partition 的第一个记录(前两行为代码,后一行为结果):

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog"), 2)
c.first
res1: String = Gnu
```

3. filter

过滤 RDD 的每一 Partition 的每一个记录(前三行为代码,后一行为结果,下面类似):

```
val a = sc.parallelize(1 to 10, 3)
val b = a.filter(_ % 2 == 0)
b.collect
res3: Array[Int] = Array(2, 4, 6, 8, 10)
```

4. filterByRange

返回指定范围内 RDD 记录,该函数只能作用于排序 RDD:

```
val randRDD = sc.parallelize(List( (2,"cat"), (6, "mouse"),(7, "cup"), (3, "book"), (4, "tv"), (1,
"screen"), (5, "heater")), 3)
val sortedRDD = randRDD.sortByKey()
sortedRDD.filterByRange(1, 3).collect
res66: Array[(Int, String)] = Array((1,screen), (2,cat), (3,book))
```

5. foreach

遍历 RDD 内每一个记录:

```
val c = sc.parallelize(List("cat", "dog", "tiger", "lion", "gnu", "crocodile", "ant", "whale", "dolphin",
"spider"), 3)
```



```
c.foreach(x => println(x + "s are yummy"))
lions are yummy
gnus are yummy
crocodiles are yummy
ants are yummy
whales are yummy
dolphins are yummy
spiders are yummy
```

6. foreachPartition

遍历 RDD 内每一个 Partition（有几个 Partition 返回几个值）：

```
val b = sc.parallelize(List(1, 2, 3, 4, 5, 6, 7, 8, 9), 3)
b.foreachPartition(x => println(x.reduce(_ + _)))
6
15
24
```

7. fullOuterJoin [Pair]

对两个 PairRDD 执行外连接：

```
val pairRDD1 = sc.parallelize(List( ("cat",2), ("cat", 5), ("book", 4),("cat", 12)))
val pairRDD2 = sc.parallelize(List( ("cat",2), ("cup", 5), ("mouse", 4),("cat", 12)))
pairRDD1.fullOuterJoin(pairRDD2).collect
```

```
res5: Array[(String, (Option[Int], Option[Int]))] = Array((book,(Some(4),None)), (mouse,(None,Some(4))),
(cup,(None,Some(5))), (cat,(Some(2),Some(2))), (cat,(Some(2),Some(12))), (cat,(Some(5),Some(2))),
(cat,(Some(5),Some(12))), (cat,(Some(12),Some(2))), (cat,(Some(12),Some(12))))
```

8. groupBy

非常重要，该函数有利于读者理解 Shuffle，请仔细研读：

```
val a = sc.parallelize(1 to 9, 3)
a.groupBy(x => { if (x % 2 == 0) "even" else "odd" }).collect
res42: Array[(String, Seq[Int])] = Array((even,ArrayBuffer(2, 4, 6, 8)), (odd,ArrayBuffer(1, 3, 5, 7,
9)))
```

```
val a = sc.parallelize(1 to 9, 3)
def myfunc(a: Int) : Int =
{
  a % 2
}
a.groupBy(myfunc).collect
res3: Array[(Int, Seq[Int])] = Array((0,ArrayBuffer(2, 4, 6, 8)), (1,ArrayBuffer(1, 3, 5, 7, 9)))
```



```

val a = sc.parallelize(1 to 9, 3)
def myfunc(a: Int) : Int =
{
  a % 2
}
a.groupBy(x => myfunc(x), 3).collect
a.groupBy(myfunc(_), 1).collect
res7: Array[(Int, Seq[Int])] = Array((0,ArrayBuffer(2, 4, 6, 8)), (1,ArrayBuffer(1, 3, 5, 7, 9)))

import org.apache.spark.Partitioner
class MyPartitioner extends Partitioner {
  def numPartitions: Int = 2
  def getPartition(key: Any): Int =
  {
    key match
    {
      case null => 0
      case key: Int => key % numPartitions
      case _ => key.hashCode % numPartitions
    }
  }
  override def equals(other: Any): Boolean =
  {
    other match
    {
      case h: MyPartitioner => true
      case _ => false
    }
  }
}
val a = sc.parallelize(1 to 9, 3)
val p = new MyPartitioner()
val b = a.groupBy((x: Int) => { x }, p)
val c = b.mapWith(i => i)((a, b) => (b, a))
c.collect
res42: Array[(Int, (Int, Seq[Int]))] = Array((0,(4,ArrayBuffer(4))), (0,(2,ArrayBuffer(2))),
(0,(6,ArrayBuffer(6))), (0,(8,ArrayBuffer(8))), (1,(9,ArrayBuffer(9))), (1,(3,ArrayBuffer(3))),
(1,(1,ArrayBuffer(1))), (1,(7,ArrayBuffer(7))), (1,(5,ArrayBuffer(5))))

```

9. groupByKey [Pair]

类似于 groupBy, 不过函数作用于 key:

```

val a = sc.parallelize(List("dog", "tiger", "lion", "cat", "spider", "eagle"), 2)
val b = a.keyBy(_.length)
b.groupByKey.collect

```

```
res11: Array[(Int, Seq[String])] = Array((4,ArrayBuffer(lion)), (6,ArrayBuffer(spider)), (3,ArrayBuffer(dog, cat)), (5,ArrayBuffer(tiger, eagle)))
```

10. histogram [Double]

计算数据直方图，第一个返回值使用均匀划分，第二个则使用用户指定的边界值：

```
val a = sc.parallelize(List(1.1, 1.2, 1.3, 2.0, 2.1, 7.4, 7.5, 7.6, 8.8, 9.0), 3)
a.histogram(5)
res11: (Array[Double], Array[Long]) = (Array(1.1, 2.68, 4.26, 5.84, 7.42, 9.0), Array(5, 0, 0, 1, 4))

val a = sc.parallelize(List(9.1, 1.0, 1.2, 2.1, 1.3, 5.0, 2.0, 2.1, 7.4, 7.5, 7.6, 8.8, 10.0, 8.9, 5.5), 3)
a.histogram(6)
res18: (Array[Double], Array[Long]) = (Array(1.0, 2.5, 4.0, 5.5, 7.0, 8.5, 10.0), Array(6, 0, 1, 1, 3, 4))
```

11. intersection

返回两个 RDD 重叠数据：

```
val x = sc.parallelize(1 to 20)
val y = sc.parallelize(10 to 30)
val z = x.intersection(y)

z.collect
res74: Array[Int] = Array(16, 12, 20, 13, 17, 14, 18, 10, 19, 15, 11)
join [Pair]
```

对两个 RDD 执行内连接：

```
val a = sc.parallelize(List("dog", "salmon", "salmon", "rat", "elephant"), 3)
val b = a.keyBy(_ .length)
val c = sc.parallelize(List("dog", "cat", "gnu", "salmon", "rabbit", "turkey", "wolf", "bear", "bee"), 3)
val d = c.keyBy(_ .length)
b.join(d).collect

res0: Array[(Int, (String, String))] = Array((6,(salmon,salmon)), (6,(salmon,rabbit)), (6,(salmon,turkey)), (6,(salmon,salmon)), (6,(salmon,rabbit)), (6,(salmon,turkey)), (3,(dog,dog)), (3,(dog,cat)), (3,(dog,gnu)), (3,(dog,bee)), (3,(rat,dog)), (3,(rat,cat)), (3,(rat,gnu)), (3,(rat,bee)))
```

12. keys [Pair]

```
val a = sc.parallelize(List((3, "dog"), (5, "tiger"), (4, "lion"), (3, "cat"), (7, "panther"), (5, "eagle")), 2)
a.keys.collect
res2: Array[Int] = Array(3, 5, 4, 3, 7, 5)
```

13. lookup

查找指定记录：

```
val a = sc.parallelize(List((3, "dog"), (5, "tiger"), (4, "lion"), (3, "cat"), (7, "panther"), (5, "eagle")), 2)
```

```
a.lookup(5)
res0: Seq[String] = WrappedArray(tiger, eagle)
```

14. max

返回 RDD 内最大记录:

```
val y = sc.parallelize(10 to 30)
y.max
res75: Int = 30
```

```
val a = sc.parallelize(List((10, "dog"), (3, "tiger"), (9, "lion"), (18, "cat")))
a.max
res6: (Int, String) = (18,cat)
```

15. mean [Double], meanApprox [Double]

计算 RDD 所有数据均值:

```
val a = sc.parallelize(List(9.1, 1.0, 1.2, 2.1, 1.3, 5.0, 2.0, 2.1, 7.4, 7.5, 7.6, 8.8, 10.0, 8.9, 5.5), 3)
a.mean
res0: Double = 5.3
```

16. persist, cache

设置 RDD 存储级别:

```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2)
c.getStorageLevel
res0: org.apache.spark.storage.StorageLevel = StorageLevel(false, false, false, false, 1)
c.cache
c.getStorageLevel
res2: org.apache.spark.storage.StorageLevel = StorageLevel(false, true, false, true, 1)
```

17. sample

根据给定的比例对数据进行采样:

```
val a = sc.parallelize(1 to 10000, 3)
a.sample(false, 0.1, 0).count
res24: Long = 960
```

```
a.sample(true, 0.3, 0).count
res25: Long = 2888
```

```
a.sample(true, 0.3, 13).count
res26: Long = 2985
```

18. saveAsTextFile、textFile

saveAsTextFile 用于将 RDD 数据持久化至 HDFS，而 textFile 则正好相反，用于从 HDFS 读取数据：

```
val a = sc.parallelize(11 to 19, 3)
a.saveAsTextFile("test/tf")
val b = sc.textFile("test/tf")
b.collect
res4: Array[String] = Array(11, 12, 13, 14, 15, 16, 17, 18, 19)
```

前两程序执行过程如图 9-47 所示，图 9-48 为在交互式执行器下，整个程序的执行过程。



图 9-47 Spark 向 HDFS 中存储文件

```
scala>
scala> val a = sc.parallelize(11 to 19, 3)
a: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:21
scala> a.saveAsTextFile("test/tf")
scala> val b = sc.textFile("test/tf")
b: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[3] at textFile at <console>:21
scala> b.collect
res1: Array[String] = Array(11, 12, 13, 14, 15, 16, 17, 18, 19)
scala>
```

图 9-48 Executor0 的 stdout

上述程序可执行的前提是，Spark 已指向 HDFS（且 HDFS 已开启），关于如何设置 Spark 指向 HDFS，请参考 9.1 节，下面是 Spark 读 HDFS 文件的另一示例，图 9-49 为程序执行过程：

```
val rawFile= sc.textFile("/user/allen/notes/notel.txt",4)
rawFile.collect

scala>
scala> val rawFile= sc.textFile("/user/allen/notes/notel.txt",4)
rawFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1] at textFile at <console>:21
scala> rawFile.collect
res0: Array[String] = Array(1 : aa bb cc dd ee ff gg, 2 : aa bb cc dd ee ff gg, 3 : aa bb cc dd ee ff gg, 4 : aa bb cc dd ee ff gg, 5 : aa bb cc dd ee ff gg, 6 : aa bb cc dd ee ff gg, 7 : aa bb cc dd ee ff gg, 8 : aa bb cc dd ee ff gg, 9 : aa bb cc dd ee ff gg, 10 : aa bb cc dd ee ff gg)
scala>
```

图 9-49 Spark 读取 HDFS 文件

19. take

返回 RDD 中设定项:

```
val b = sc.parallelize(List("dog", "cat", "ape", "salmon", "gnu"), 2)
b.take(2)
res18: Array[String] = Array(dog, cat)
```

20. union, ++

对两个集合执行并操作:

```
val a = sc.parallelize(1 to 3, 1)
val b = sc.parallelize(5 to 7, 1)
(a ++ b).collect
res0: Array[Int] = Array(1, 2, 3, 5, 6, 7)
```

9.5 实战 Spark 之 WordCount

作为 M-S-R 范式经典示例, WordCount 能够完整诠释 Map、Shuffle 和 Reduce 整个过程, 下面是 Spark 版的 WordCount 代码^[3], 用户可使用 “IntelliJ IDEA” 编写、编译和打包:

```
package edu.njupt
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
object WordCount {
  def main(args: Array[String]) {
    val conf = new SparkConf()
    val sc = new SparkContext(conf)
    val line = sc.textFile(args(0))
    //打印至控制台
    line.flatMap(_.split(" ")).map(_._1).reduceByKey(_+_).collect().foreach(println)
    存储至 HDFS
    //line.flatMap(_.split(" ")).map(_._1).reduceByKey(_+_).saveAsTextFile(args(1))
    sc.stop()
  }
}
```

假定打包好的文件名为 spkAction.jar, 存于 iclient0 机 “/home/allen” 目录下, 则可使用下述命令向 Spark 提交本应用:

```
[allen@iclient0 spark-1.5.2-bin-hadoop2.6]$ bin/spark-submit --master spark://cmaster0:7077 --class
njupt.WordCount ~/spkAction.jar
```

当然，用户也可以使用下述命令，进入交互式执行器后，一句一句执行：

```
[allen@iclient0 spark-1.5.2-bin-Hadoop2.6]$ bin/spark-shell --master spark://cmaster0:7077
```

下面也是 WordCount 的 Spark 版代码，请读者打包调试：

```
object WordCount{
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf()
    val sc: SparkContext = new SparkContext(conf)
    val rawFile = sc.textFile(args(0))
    val partitions: Array[Partition] = rawFile.partitions
    rawFile.flatMap(line => line.split(" ")).map(word => (word, 1)).groupByKey().map((P: (String,
    Iterable[Int])) => (P._1, P._2.sum)).saveAsTextFile(args(1))
    sc.stop()
  }
}
```

虽然一个版本的 WordCount 代码比上一稍显复杂，但实际上，这两个代码功能是一样的，从理解角度上说，第二个 WordCount 代码更利于理解。稍加分析，可以看出：

```
reduceByKey()
等同于
groupByKey().map()
```

9.6 实战 Spark 之 MLlib

关于 MLlib 模块，Spark 官方文档（如下网址）有清晰说明，编者给出的 SVM 代码也来自该网址：

```
http://spark.apache.org/docs/latest/
```

进入该网址后，依次点击“MLlib”、“SVM”即可找到 SVM 源码。若网址打不开，可在 Baidu、Bing、Google 搜索“spark”，待进入 Spark 官网后，依次点击“Documentation”、“Latest Release”、“MLlib”、“SVM”。

SVM 是一种常见的分类策略，Spark 中的并行 SVM 可在大规模数据集上实现快速训练。

（1）优化目标

Spark 中的并行 SVM 是一个线性 SVM，其优化目标和损失函数分别为：

$$\min_w f(w) := \lambda R(w) + \frac{1}{n} \sum_{i=1}^n L(w; x_i, y_i)$$

$$L(w; x, y) := \max\{0, 1 - yw^T x\}$$

上述中的 $f(w)$ 为线性 SVM 的优化目标， $L(w; x, y)$ 为损失函数，该模型常用于大

规模分类任务。默认情况下, Spark 上的并行 SVM 采用 L2 正则化来训练模型, 当然其也支持 L1 正则化, 不过在此种情况下, 优化问题将退化为线性规划。

求出上述优化问题中的法向量 w , 此时, 当给定一个新的样本点 x 时, 模型将依据 $w^T x$ 的结果来做出预测。默认情况下, 如果 $w^T x \geq 0$ 则认为其为正类, 反之则将其分为负类。

(2) 优化策略

对于优化问题 $\min_w f(w)$, 可采用多种算法求解。不过, 由于 Spark 平台的特殊性, 其目前仅支持随即梯度下降法 (Stochastic Gradient Descent, SGD) 和内存受限拟牛顿法 (Limited-memory BFGS, L-BSGF) 两种优化策略, 下面仅介绍 SGD 优化策略。

首先, 对目标函数 $\min_w f(w)$ 求偏导, 得到其梯度:

$$f'_{w,i} := L'_{w,i} + \lambda R'_w$$

根据给定的学习速率为 γ , 可得到 w 的更新公式:

$$w^{(t+1)} := w^{(t)} - \gamma f'_{w,i}$$

综上, 按照 SGD 官方文档, 可知 SGD 执行算法如下:

SGD 算法过程:

Input: $train(m := |train|), T, \gamma, \omega_1$

Output: ω_T

$t := 1, error := \inf$

while $t \leq T$ or $error < \epsilon$ do

 Draw $x \in \{train\}$ uniformly at random

$w := w - \gamma f'$

$error := \frac{1}{m} L(w; x, y), t := t + 1$

end for

return w

(3) SGD 并行策略

由于 SGD 仅需要部分或单个样本即可实现更新 w , 因此 SGD 支持并行化操作。实际操作中, 其并行策略为: 将大规模样本集合拆成多个子样本集合, 然后在各子样本集上执行 SGD 得到各自的模型, 最后将所有模型按某种策略合并起来得到最终模型。假定现有 $m \cdot k$ 条训练样本, Spark 集群中有 k 台 Worker, 则整个并行处理过程可描述如下 (图 9-50):

Step0 开始, 初始化数据集, Spark 集群

Step1 将训练数据集均匀分为 k 份, Spark 中每个 Worker 机持有一份数据

Step2 各 Spark 中的 Worker 机使用上述的 SGD 算法处理本 Worker 所持有数据

Step3 将 Step2 中各 Worker 机计算所得的 w 聚合起来, 依据公式 $w = \frac{1}{k} \sum_{i=1}^k w_{i,t}$ 计

算出最终 w

Step4 得出最终 w ，结束

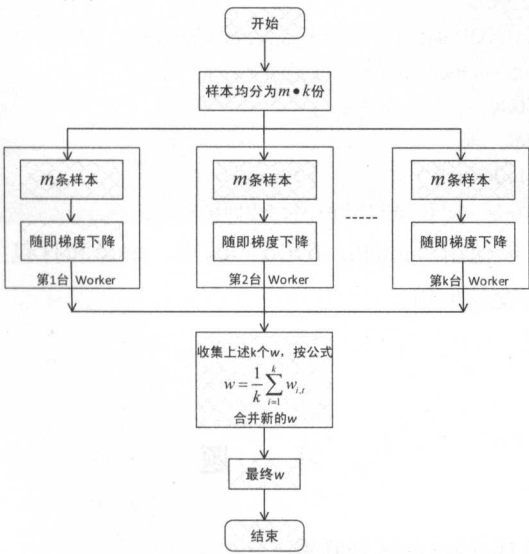


图 9-50 Spark 上并行 SVM 执行过程图

上述操作中，步骤 Step2 采用的算法即是本节第二部分讲述的 SGD 算法，由于 SGD 算法仅依赖部分或单个样本，无需要遍历全局数据，因此可使用 SGD 实现并行化。

(4) 开发过程

SVM 算法训练集和测试集均来自其解压后的目录 “spark-1.5.0-bin-hadoop2.6/data/mllib/sample_libsvm_data.txt”，用户须将该文件上传至 HDFS，接着使用交互式执行器，依次执行下述代码：

```
import org.apache.spark.mllib.classification.{SVMModel, SVMWithSGD}
import org.apache.spark.mllib.evaluation.BinaryClassificationMetrics
import org.apache.spark.mllib.util.MLUtils
// Load training data in LIBSVM format.
val data = MLUtils.loadLibSVMFile(sc, "data/mllib/sample_libsvm_data.txt")
// Split data into training (60%) and test (40%).
val splits = data.randomSplit(Array(0.6, 0.4), seed = 11L)
val training = splits(0).cache()
val test = splits(1)
// Run training algorithm to build the model
val numIterations = 100
val model = SVMWithSGD.train(training, numIterations)
// Clear the default threshold.
model.clearThreshold()
// Compute raw scores on the test set.
val scoreAndLabels = test.map { point =>
    val score = model.predict(point.features)
```

```
(score, point.label)
}
// Get evaluation metrics.
val metrics = new BinaryClassificationMetrics(scoreAndLabels)
val auROC = metrics.areaUnderROC()
println("Area under ROC = " + auROC)
// Save and load model
model.save(sc, "myModelPath")
val sameModel = SVMModel.load(sc, "myModelPath")
```

当然, 用户也可以使用 “IntelliJ IDEA” 编写、编译和打包上述代码, 接着使用 “spark-submit” 命令提交该应用。

习 题

1. 简述 Spark 和 Hadoop 的区别联系。
2. 既然 MapReduce 框架已经实现了 M-S-R 范式, 为何又开发 Spark 来实现 M-S-R 范式?
3. 简述 Spark 框架功能作用及其体系架构。
4. 简述手工部署 Spark、使用 Ambari 部署 Spark 的步骤。
5. 简述 Spark 访问接口。
6. 在 IntelliJ IDEA 环境下, 请分别使用 sbt 和不使用 sbt 搭建 Spark 开发环境。
7. 简述 Spark-App 编程步骤。
8. 简述在 Standalone 模式下和 YARN 模式下, Spark-App 执行步骤。
9. 在大型系统中, 如何使用 Spark 来提供交互式服务、在线服务和离线服务?
10. 在大型系统中, 如何使用 Spark 来处理增量数据?
11. 常见的 Spark-App 调优技术。
12. Spark 的计算模型以及 RDD 是什么?
13. 请编写代码, 证明 Spark 的确是在并行处理用户数据。

参考文献

- [1] <https://amplab.cs.berkeley.edu/>
- [2] <http://spark.apache.org/>
- [3] <http://spark.apache.org/docs/latest/quick-start.html>

第10章

数据流实时处理系统 Storm

HADOOP

BEING DIGITAL



Storm 是一个基于数据流的实时处理系统,当数据到达系统后,会立刻被 Storm 系统载入到实时处理流中并在很短的时间内完成处理。本章首先讲解 Storm 基础理论,接着讲述其典型应用。

10.1 Storm 简介

Storm^[1]是由 BackType 开发的实时数据分析软件,旨在分析一个组织在 Twitter 发布的消息的影响力、分析 Twitter 消息被他人重复转发的频率。Twitter 于 2011 年 7 月正式收购了 BackType 并在同年 8 月将 Storm 开源。目前 Storm 已发展成为一套完整大数据实时处理解决方案,本节从理论方面扼要讲解 Storm 体系架构,为 Storm 实战提供理论支持。

10.1.1 与 Hadoop 的关系

实际上,Storm 和 Hadoop 并无可比性,Storm 用于实时计算领域,Hadoop 常用于离线分析和数据持久化。对于如下相同场景(参考自《Storm 分布式实时计算模式》),当分别使用 Hadoop 和 Storm 处理时,虽然最终的效果相同,但时间开销完全是天壤之别。

1. 场景

现有一组应用程序“应用 1”、“应用 2”和“应用 3”,它们可以是门户网站、微博等,这些应用的后台都使用了 logback (<http://logback.ch>) 将结构化的日志消息写入磁盘(访问记录、错误消息等)。现有如下两种方式处理这些日志。

2. 离线方式(Hadoop 方式)

当采用 Hadoop 来分析这些日志时,常规方法是,使用 Flume 将这些日志全部都导入 HDFS,然后使用 Hive、Pig 或 MapReduce 从数据中提出有用信息(图 10-1)。

此时,由于 MapReduce 是个批处理机制,其整个处理的时间明显减慢了我们的反应时间,从 log 数据中找出模式的时间往往以小时或天来计,一个特定的事件发生后,可能已经来不及响应了。如果在模式出现时能积极感知,而不是事后发现,这是非常令人期待的事情。

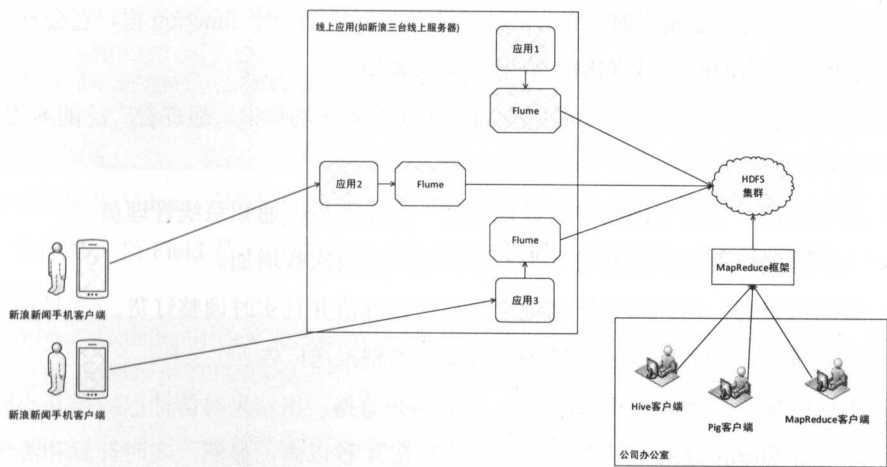


图 10-1 离线分析示例图

3. 实时方式（Storm 方式）

当采用 Storm 实时处理机制时,Kafka Appender 会将数据实时发往 Kafka 队列,Kafka Spout 则从队列中取出消息并发往 Storm Topology, Topology 会对消息进行实时处理（如过滤），并将一些敏感信息实时发往 XMPP，接着 XMPP 会自动通知前端用户（图 10-2）。

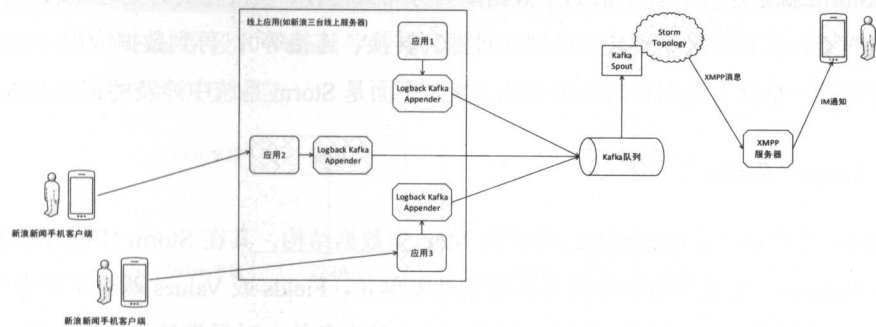


图 10-2 实时分析示意图

上述场景中涉及 Logback、Kafka、Storm 和 XMPP。其中，Logback 是一个日志记录框架，用于以标准格式记录服务器产生的日志数据；Apache Kafka 是一个开源分布式“发布-订阅”消息系统，其为高吞吐、持久化实时数据进行了特别设计和优化，和 Storm 类似，Kafka 设计足以支撑商业软件的大规模水平扩展，支持每秒数十万消息的处理。XMPP（Extensible Messaging and Presence Protocol）是一种基于 XML 的标准，用来进行即时通信、展示信息和通信录维护的服务软件。

Kafka spout 从 Kafka 队列中读取数据，并且发射到 Storm Topology 中，而该 Topology 由一系列内置和自定义的 Trident 组件（functions、filters、state 等）构成，检测数据流中

的模式。当监测到特定模式时, Topology 会发送 Tuple 到一个 function 里, 它会发送一个 XMPP 服务, 使用即时通信 (IM) 给用户发送通知。

显然, 上述场景代表了一个通用主题, 在众多商业场景中, 都有着广泛的应用空间, 包括下述应用。

- 应用监控: 例如当网络中的错误到达一定比率后, 通知系统管理员。
- 入侵检测: 例如检测异常行为, 登录尝试失败次数增加。
- 供应链管理: 例如用来检测特定商品销售峰值并且实时调整订货。
- 在线推荐: 例如发现流行趋势并且动态调整推送广告。

对于这些场景, 如果使用 Hadoop 分析这些数据, 出结果时估计已经是几小时后的事了, 而使用 Storm 处理, 整个过程可以控制在 4 秒以内, 显然, 实时计算和离线计算目的虽相同, 效果却完全不同。

10.1.2 基础概念

对大部分编程人员来说, 若不借助数据库与分布式内存, 很难实现海量数据的实时处理。Storm 就是这样一套不借助于数据库与分布式内存、仅依托实时处理流的大数据实时处理框架。从数据采集到数据处理 (过滤、转换、连接等), 再到数据应用, Storm 为我们提供了一整套大数据实时处理解决方案, 下面是 Storm 系统中涉及的诸多基本概念。

1. Tuple (元组)

Tuple 实质是一个 <key,value> 形式的 Map 型数据结构, 其在 Storm 中的专业表述为 <Fields,Values>, 它是 Storm 中消息传递的基本单元。Fields 或 Values 两个字段本身可以是任意复杂数据结构, 不过应满足可序列化这一基本条件。对于常见的基本类型, Storm 都有相应序列化版本。

值得注意的是, 在实际使用时, 用户只需要按序填充 Values 字段, 无须关心 Fields 字段。这是由于各个组件间传递的字段名称已经事先定义好, 在 Storm 计算模型章节, 编者将再次讲述 Tuple 数据结构。

例 1 请给出两个 Tuple 实例^[2]。

解① Tuple 是 Storm 组件间交换数据时的基本单元, TupleImpl 类是 Tuple 中最常用子类, 下面的代码首先实例化一个 TupleImpl 实例, 接着设置此实例字段名、字段值, 并分别做了输出。

```
TupleImpl iTupleImplA=new TupleImpl();    //实例化一个 Tuple
iTupleImplA.setFields(new Fields("xxxA")); //命令此 iTupleImpl 的 Fields 名 xxx
```



```
iTupleImplA.getFields(); //获取 iTupleImpl 当前 Fields 的命名
输出: xxxA
iTupleImplA.setValues(new Values("222A")); //设置 xxx 字段的值为 222
iTupleImplA.getStringByField("xxxA"); : //获取 xxx 字段对应值（即 222）
输出: 222A
```

②Fields 和 Values 字段非常灵活,可以是一个序列,也可以是复合类型。下面的 Fields 就是一个序列,当 Field 是一个序列时,Values 也要是一个序列,其中每个值和 Fields 每个字段相对应。

```
TupleImpl iTupleImplB=new TupleImpl(); //实例化一个 Tuple
//设置此 iTupleImpl 的 Fields 包含三个 Field, 分别为 xxxB、yyyB、zzzB
iTupleImplB.setFields(new Fields("xxxB","yyyB","zzzB"));
iTupleImplB.getFields(); //获取 iTupleImpl 当前 Fields 的命名
输出: [xxxB, yyyB ,zzzB]
//设置 xxxB 字段的值为 222B, yyyB 值为 Int 型 33, zzzB 字段值为###B
iTupleImplB.setValues(new Values("222B",33,"###B"));
iTupleImplB.getStringByField("xxxB"); //获取字段 xxxB 对应的值
输出: 222B
iTupleImplB.getIntField("yyyB"); //获取字段 yyyB 对应的值
输出: 33
iTupleImplB.getStringByField("zzzB"); //获取字段 zzzB 对应的值
输出: ###B
```

图 10-3 为 Fields 字段和 Values 字段对应关系图,从图中可以看出,Fields 声明的序列和 Values 值的序列相互对应。

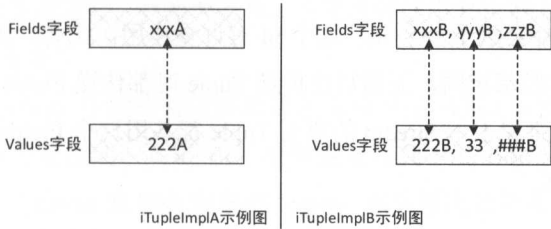


图 10-3 TupleImpl 数据结构示意图

上述是编者虚构的两段代码,除非在真实的 Storm 上下文环境中,否则上述代码不可执行,借助上述代码,读者须明白,Tuple 就是这样的一个<Fields,Values>形数据结构。

2. Stream（流）

Stream 是 Storm 的实时处理功能的核心抽象体,是一个无界的 Tuple 序列,源源不断的 Tuple 就组成了 Stream。有 Stream 就一定有源头和处理流的水坝,Storm 分别称 Stream 源和水坝为 Spout 和 Bolt。

Spout 是流的源头,通常从外部数据源读取数据并转化为 Tuple,然后转发到各个 Bolt

中；Bolt 是流处理节点，它会处理流向本 Bolt 的所有 Tuple，常见的处理操作是过滤、join、连接数据库等。

正如图 10-4 所示，称顶点到顶点之间的数据流为 Stream，称数据源为 Spout，称流处理节点为 Bolt，称由 Spout、Stream、Bolt 构成的图为 Topology。Spout 可以将 Tuple 发射到一个或多个 Bolt，同样 Bolt 可以订阅一个或多个 Spout，特别地，Bolt 还能够订阅一个或多个上层 Bolt，即 Topology 中可以有多 Spout 和多层 Bolt。

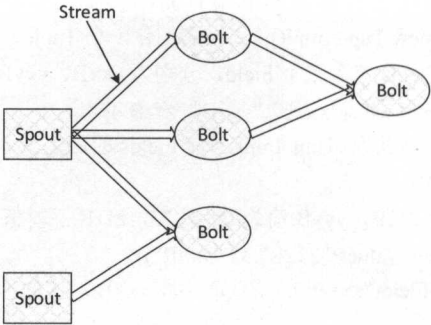


图 10-4 Storm 流示例图

Topology 节点之间的链接表示 Tuple 应该如何传递，比如，如果从 SpoutA 到 BoltB 有一个链接，从 SpoutA 到 BoltC 有一个链接，从 BoltB 到 BoltC 有一个链接，那么每次 SpoutA 将同时发送 Tuple 到 BoltB 和 BoltC，BoltC 也将收到 BoltB 发送的 Tuple。

当 Spout 向 Bolt 发送 Tuple 时，实际上 Fields 字段是相同的，不必每次都发送。每当 Stream 被申明后都会赋予一个 id。这个 id 有许多作用，比如，因为单个 Stream 传递的 Tuple 里 Fields 字段都相同，无须每次传递 Tuple 时都传递 Fields，故 Storm 赋予一个 Stream 一个 Fields，隶属于本 Stream 的所有 Tuple 都共用这个 Fields，从而避免不必要的传递，节省网络带宽。

3. Spout（喷口）

Spout 是整个 Topology 的 Stream 的来源，是一个 Topology 中产生数据流的组件。目前 Spout 支持读取的数据源包括 Kafka、HDFS、HBase、Hive、JDBC、Redis、Solr、Azure EventHubs、Flux。Storm 框架会不停地调用 Spout 里的 nextTuple() 来实时读取这些输入源（Kafka、HDFS 等）中的数据，除非手动关闭整个 Topology，否则永不停止。

Spout 在其 nextTuple() 方法里将原始数据转换为 Tuple 后随即将这个 Tuple 发出，为保证此 Tuple 被下层 Bolt 正确处理，Spout 还提供了 ack() 与 fail 方法，下一层获取 Tuple 的地方响应该 Tuple 即可确认本 Tuple 已经传送成功（图 10-5）。

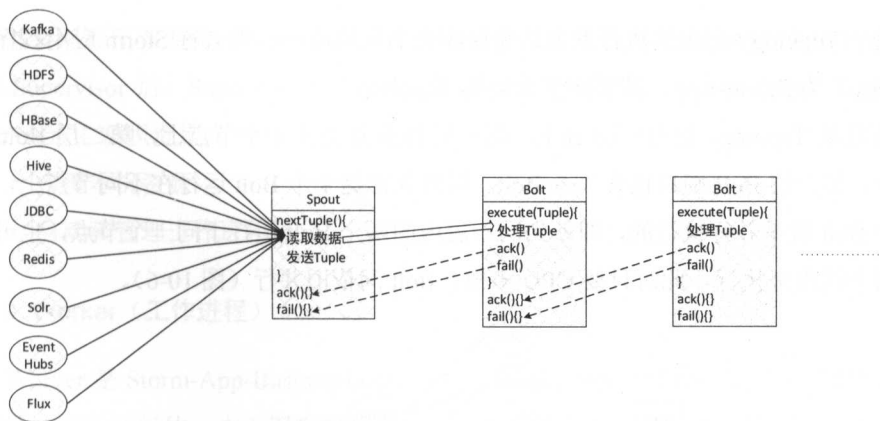


图 10-5 Spout 内部方法及其数据源示例图

需要说明的是不论是 Spout 还是 Bolt，它们都可以定义成可靠型和非可靠型，当用户需要使用可靠的 Spout (Bolt) 时手动开启 `ack()` 和 `fail()` 即可，当需要使用非可靠的 Spout (Bolt) 时不实现该方法即可；当 Topology 里 Stream 太深（即节点层次太深）时，还可定义跨节点 `ack()` 和 `fail()`。

4. Bolt（螺栓）

一般情况下，开发人员会将 Spout 部署在生产机上，以实时读取线上服务生成的各种数据。此时为保障生产机时刻拥有充足处理能力，应确保 Spout 尽量轻量级，故 Spout 中正常不会定义诸如数据过滤、转换等操作，而是选择使用 Bolt 模块，来处理 Tuple（过滤、Join、访问数据库等）。

可以说，在 Topology 中的所有处理都是在 Bolt 中完成的，Bolt 是流的处理节点。Bolt 也可以定义成多个层次的，上层 Bolt 将数据发送到下层 Bolt 中。Bolt 通过其中的 `execute()` 方法获取上层 Spout 或 Bolt 发来的 Tuple，然后再次转换成 Tuple 后发往下一层 Bolt，直至 Bolt 层次结束。默认，Bolt 只会发出一个 Stream，当需要 Bolt 发出多个 Stream 时，须调用 `declareStream()` 方法。

5. Topology（拓扑）

Topology 是 Storm 中运行的一个实时应用程序，其是由 Spout、Bolt、Stream 构成的 DAG 图，因为各个组件间的 Tuple 流动而形成了逻辑上的 Topology 结构，故称为 Topology。

Storm 称其应用程序为 Topology，一个 Topology 类似于一个 MapReduce 作业。不同的是，MapReduce 作业总有个时间期限，而一旦提交一个 Topology，除非手工将其停止，

否则这个 Topology 会永远执行下去。为保障全书风格统一，编者称 Storm 应用程序（即 Topology）为 Storm-App，其等同于此处的 Topology。

假设某 Topology 具有三层 Bolt，第一层 Bolt 运行在十个节点上，第二层 Bolt 为五个节点，第三层 Bolt 则只包含 3 个 Bolt。只要保证这十个 Bolt 运行在不同节点上，那么这十个 Bolt 就是并行执行的，即使当同一层内的两个 Bolt 运行于同一个节点，也可以设定用两个线程来执行，此时只要 CPU 多核，Bolt 间依旧并行（图 10-6）。

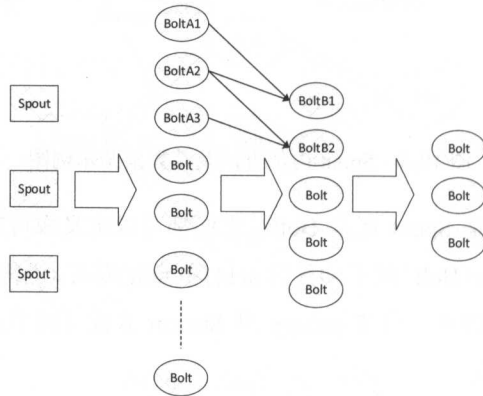


图 10-6 Topology 实例

6. Stream grouping（流分组）

通过 Stream grouping，可以为每个 Bolt 指定输入流。通过流分组，可以要求 BoltB1 只接收 BoltA1、BoltA2 的输入流，同样也可以指定 BoltB2 只接收 BoltA2、BoltA3 的输入流。Storm 内置了 7 种流分组方式，可完全满足用户各类需求。

7. Nimbus（主进程）

Storm 集群采用 master/slave 架构，主节点运行主服务 Nimbus 进程，从节点运行从服务 Supervisor 进程。

在大部分 master/slave 架构的框架中，master 都是负责资源管理、任务响应类任务，Nimbus 也不例外，在 Storm 系统中，其主要负责：

- 管理集群资源
- 接收 Storm-App
- 启动 Storm-App

8. Supervisor（从进程）

整个集群内，除了主节点外，其他从节点都要部署从服务 Supervisor，和 Nimbus 充

当集群级别管理者不同, Supervisor 只管理本机资源, 此外当 Nimbus 将 Storm-App 下派到本 Supervisor 后, Supervisor 还负责启动 Worker 进程来执行 Storm-App。总的来说, Supervisor 主要职责是:

- 管理本机资源
- 启动和管理工作进程

9. Worker (工作进程)

Worker 是 Storm-App-BusinessLogic (用户编写的 Storm 代码, 实现了需要处理的业务逻辑) 实际运行体, 由于用户编写 Storm-App 时就是编写 Spout 和 Bolt, 故 Worker 是 Spout 或 Bolt 的实际执行体。一般情况下, 每个 Storm-App 都包含多个处理流, 也就是多个 Worker, 每个 Worker 可以包含多个 Spout 和 Bolt。Worker 进程或 Spout 或 Bolt 数量对应关系请参见 Executor。

10. Task (任务)

逻辑概念, 一个具体的 Spout 或 Bolt 称为一个 Task。

11. Executor (执行器)

物理概念, 执行 Spout 或 Bolt 的线程。前文说由 Worker 进程来执行 Spout/Bolt, 实际上, Spout/Bolt 的真实执行者, 是隶属于 Worker 进程的众多线程。Spout/Bolt 与 Executor 对应关系是: 一个 Executor (线程) 可以做一个多个 Task (Spout、Bolt 实体), 但不支持多个 Executor 同时做一个 Task。显然一个 Worker 可以拥有多个 Executor (线程), 故一个 Worker 内可运行多个 Spout/Bolt。

12. Reliability (可靠性)

Storm-App 的可靠性包含两层含义, 一是确保 Tuple 不丢失, 二是确保正在执行任务的 Worker 进程崩溃后, 系统能够重新调度此 Worker。

Storm 通过追踪每一个 Tuple 来确保该 Tuple 不会丢失, 上文讲述的 ack()、fail() 就是此机制的具体体现, 此外, 还可对每一 Tuple 设置“超时设置”, 超时的 Tuple 会被重发。

当执行 Worker 的物理机器宕机后, Nimbus 会在另一机器上再次启动该 Worker, 并将之前发送的未应答 Tuple 再次发送。借助 ZooKeeper 集群, 可为集群本身配置三个以上的 Nimbus, 从而解决 Nimbus 自身宕机的风险。

10.1.3 体系架构^[3]

Storm 共有两层体系架构，第一层为采用 master/slave 架构的集群管理器，第二层为 DAG 流式处理器，第一层资源管理器主要负责管理集群资源、响应和调度用户任务，第二层流式处理器则实际执行用户任务。

1. 集群资源管理层

Storm 的集群资源管理器采用 master/slave 架构，主节点上部署主服务 Nimbus，从节点上部署从服务 Supervisor（图 10-7）。

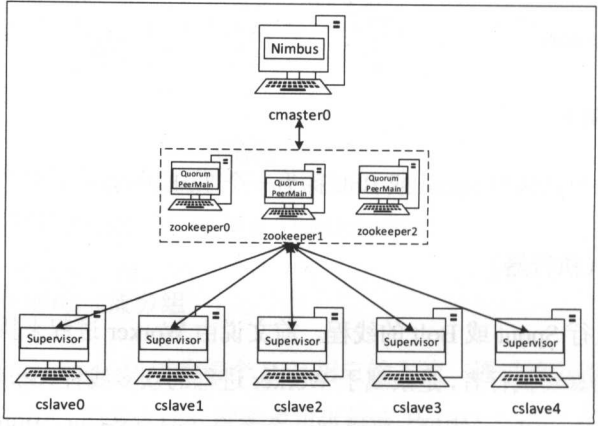


图 10-7 Storm 资源管理层体系架构

称集群信息（Nimbus 协议、Supervisor 节点位置）、任务分配信息等关键数据为元数据。Storm 使用 ZooKeeper 集群来共享元数据，而这些元数据对 Storm 非常重要，比如 Nimbus 通过这些元数据感知 Supervisor 节点，Supervisor 通过 ZooKeeper 集群感知任务分配情况。

(1) Nimbus

Storm 资源管理主服务、任务调度主服务，部署在主节点上，主要负责：

- 管理集群资源
- 接收 Storm-App
- 启动 Storm-App

(2) Supervisor

Storm 资源管理从服务，集群中每个从节点上都须部署该服务，主要负责：

- 管理本机资源
- 启动和监管 Worker 进程

(3) ZooKeeper 集群

Storm 使用 ZooKeeper 来协调集群和任务状态信息，其主要为 Storm 提供如下服务：

- 协调集群状态信息
- 协调任务分配信息

Storm 使用 ZooKeeper 来协调集群状态信息，比如 Supervisor 之间的 Nimbus 拓扑度量。当在 Storm 集群上运行 Storm-App 时，Storm 使用 ZooKeeper 来协调任务分配信息，比如任务的当前分配状况、工作节点 Worker 进程状态等。

Storm 对 ZooKeeper 的使用相对比较轻量化，不会造成过多的资源负担，这是因为对于重要级的数据传输操作，比如发布 Storm-App 时的 Jar 包传输，实际上底层使用 Thirft 进行通信，各个 Topology 节点之间的数据传输则采用 Netty 或 ZMQ。

2. Storm-App 任务执行层

Storm-App 实质上是一个由处理节点和流构成的 Topology，且整个 Topology 过程本身就是一个处理流，比如 SpoutA 只需要按定义将数据发往 BoltA 即可，两者之间无领导与被领导关系，故无需 master/slave 架构，采用对等结构即可，即集群中每台机器上都启动 Worker 进程（图 10-8）。

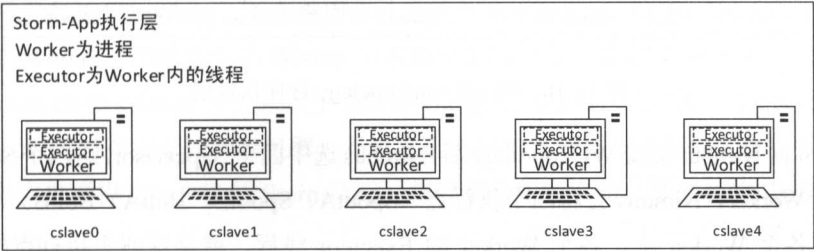


图 10-8 Worker 示意图

当 Storm 集群正常运行时，集群中只有 Nimbus、Supervisor 和 ZooKeeper 这三个实体，Worker 并不存在。当 Client 向 Nimbus 提交 Storm-App 时，Nimbus 才会要求（被 Nimbus 调度算法选中的）Supervisor 启动 Worker 进程，Worker 进程内部会启动一系列线程，Storm 称一个线程为一个 Executor，这些（Executor）才是 Storm-App 中用户编写的 Spout、Bolt 实际执行者。

以一个包含两个输入 Spout、一个单词拆分 Bolt 和一个单词统计 Bolt 的 WordCountTopology 为例，下面讲述其 Topology 和执行状态图。

WCTopology 包含两个 Spout，一个 BoltA 和一个 BoltB，按道理，BoltA 应当订阅来自 SpoutA 和 SpoutB 两个流，故其 Topology 可描述如下（图 10-9）。

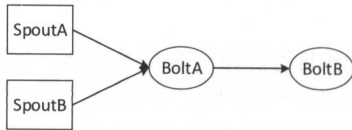


图 10-9 WCTopology 任务 Topology

当 Client 提交 WCTopology 执行时，Nimbus 会选中一系列 Supervisor，这些 Supervisor 会各自启动 Worker 进程，这些 Worker 进程才是 WCTopology 的实际执行进程，不过这些 Worker 之间没有领导与被领导关系，事实上它们地位相等，是一个对等结构。Worker 进程在启动后，会启动一系列附属线程（称 Executor），这些 Executor 会来执行用户编写的 Spout 或 Bolt 模块。

图 10-10 所示的 Topology 有四个执行点，假定设定了四个 Worker 来执行该 Topology，且不再进行任何线程优化，则 WCTopology 的执行层默认执行状况如下。

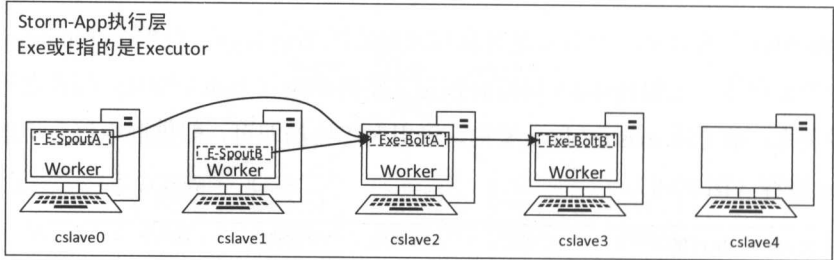


图 10-10 WordCountTopology 程序执行层

Client 向 Nimbus 提交 WCTopology 后，Nimbus 选中四个 Supervisor，这四个 Supervisor 随即启动 Worker，Nimbus 会将四个执行点（SpoutA、SpoutB、BoltA、BoltB）尽量均匀地分散到各个 Worker 上，各个 Worker 的 Executor 线程，就是这四个执行点的实际执行体。

3. Topology 执行过程

当 Client 需要在 Storm 集群上运行 Topology（Storm-App）时，Nimbus 会将 Topology 本身的相关信息、任务分配信息写入 ZooKeeper 共享存储器，各个 Supervisor 会定期扫描该存储目录，当获知本 Supervisor 有任务时，该 Supervisor 会启动 Worker 来执行用户 Topology。

当 Client 向集群提交 Topology 时，Nimbus（通过 Supervisor）会启动一系列 Worker 来执行用户 Topology。不过该 Topology 执行点（用户编写的 Spout 和 Bolt 实例）和 Worker

之间无直接对应关系，系统默认是将执行点总数均匀分散到各个 Worker 上执行，各个 Worker 根据系统分配下来的 Task 数，启动对应数量的 Executor 线程来执行这些 Task。

需要注意的是，在 Topology 运行过程中，这一系列 Worker 进程只属于本 Topology，各 Worker 里面的 Executor 线程只会执行本 Topology 的 Spout、Bolt 实例。比如，此时若有另一用户向 Storm 集群提交了一个新的 Topology，Storm 集群同样也会为此 Topology 启动一系列 Worker，这些 Worker 和之前的 Worker 甚至可以在相同的机器上，但它们一定不是之前的 Worker。

下面以 WCTopology 和 TopNTopology 为例，讲述 Storm 集群中，这两个 Topology 执行过程。

假定 iclient0 提交了 WCTopology，而 iclient1 提交了 TopNTopology，且这两个 Topology 的执行序列如图 10-11 所示，则正在运行这两个 Topology 的 Storm 状态图可绘制成图 10-12。

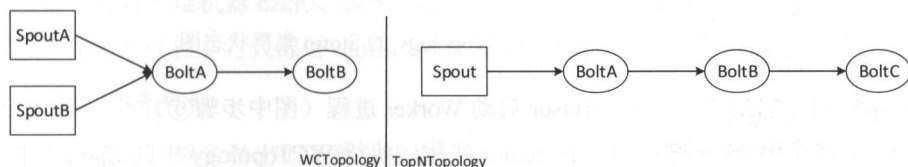


图 10-11 WCTopology 和 TopNTopology 执行序列图

图 10-12 中从 cslave0~cslave3, WCTopology 共启动了四个 Worker 进程，这四个 Worker 进程专属于本 WCTopology，不会再用来计算其他 Storm-App。恰巧，TopNTopology 也启动了四个 Worker，不过这四个 Worker 中有两个在同一台机器上（cslave4），另外两个则分别在 cslave3 和 cslave1 上。隶属于 WCTopology 的 Worker 和隶属于 TopNTopology 的 Worker 可以同时存在于同一台机器，不过它们之间相互隔离，无任何关系。读者需要注意，每当向集群提交一个新的 Storm-App，集群都会相应启动一系列新的 Worker 来执行该 Storm-App。

当 Storm 集群未运行任何 Storm-App 时，集群中只存在 Nimbus、Supervisor 和 ZooKeeper 三大实体。当 Client 需要向 Storm 集群上提交 Topology 时，首先其会向 Nimbus 提交 Storm-App，接着，Nimbus 启动一系列 Worker 来执行该 Storm-App。当 Storm 集群正在执行 Storm-App 时，实际上是集群中一系列 Worker 执行该 Storm-App。下面以 WCTopology 为例，讲述提交、启动、执行整个过程。

Step1 iclient0 向 Nimbus 提交 WCTopology（图中①）。

Step2 Nimbus 根据 EvenSchedule 调度算法，对 WCTopology 进行资源分配（步骤②）。

Step3 Nimbus 将对 WCTopology 的任务调度信息写入 ZooKeeper（图中步骤③）。

Step4 各个 Supervisor 定期访问 ZooKeeper 存储器时，读取到了 Nimbus 分配给本 Supervisor 的任务（图中步骤④）。

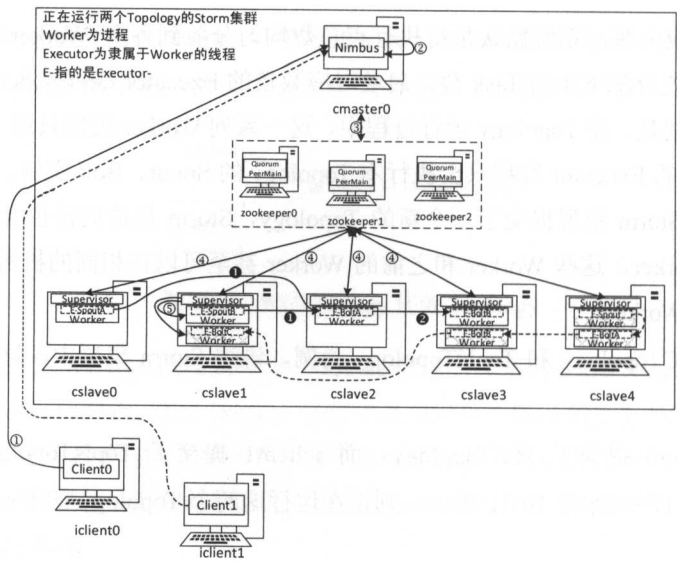


图 10-12 正在运行两个 Topology 的 Storm 集群状态图

Step5 被分配到任务的 Supervisor 启动 Worker 进程（图中步骤⑤）。

Step6 各个 Worker 进程启动 Executor 线程，执行 WCTopology 中的 SpoutA 和 Bolt（图中实现连接的各个隶属于 WCTopology 的 Spout 和 Bolt 实体）。

Step7 除非管理员显式关闭 WCTopology，否则 WCTopology 一直运行下去，永不停止。

TopNTopology 的执行过程和上述过程相似，请读者务必注意当集群中出现一个新的 Storm-App 时，Nimbus 会启动一系列新的 Worker 来执行这个该应用，而不是使用正在执行的 Storm-App 的 Worker。

10.1.4 集群部署

作为一个强大的实时分布式框架，Storm 有其独立的运行平台^[4]，在该平台下，Storm 运行稳定，性能卓越。不过，在当前大数据领域，工具纷繁复杂，数不胜数，各类组织机构都在试图将这些工具整合到一个平台上，而在这类平台中，最主流的当属 YARN，Storm 集群也支持 YARN 部署。

1. 独立部署

当采用独立方式部署 Storm 集群时，由于 Storm 依赖 ZooKeeper，故在部署 Storm 之前还要先部署 ZooKeeper 集群。Storm 本身部署非常简单，将其解压，配置 ZooKeeper 地址、配置 Nimbus 地址即可，总的来说，可按下述完整步骤部署 Storm 集群。

Step1 制定部署规划。

Step2 准备硬件机器，准备机器操作系统环境，准备机器网络环境。

Step3 对集群内每一台机器，修改机器名，关闭防火墙，安装 jdk。

Step4 为每台机器，添加集群级别域名映射。

Step5 部署 ZooKeeper 集群。

Step6 主节点解压 Storm，配置 Storm。

Step7 将配置好的 Storm 复制至所有 slave 机。

Step8 启动 Storm。

Step9 测试 Storm。

显然，在上述实际部署过程中，各机并未分角色部署不同程序包，而是所有机器都部署相同的 Storm 包。不过，在启动 Spark 时，各机则分不同角色启动不同进程。先给出如下场景，编者在此场景下，一步步部署 Storm。

场景：现有一堆机器 czkr0、czkr1、czkr2，cmaster0，cslave0~N，iclient0、iclient1，请在这些机器上以独立方式部署 Storm 集群。根据上述过程，可操作如下：

1) 制定部署规划

根据机器名，编者做出如下部署规划。

①ZooKeeper 集群：czkr0、czkr1、czkr2 部署 ZooKeeper。

②Storm: cmaster0 机充当主节点，部署 Nimbus 服务；cslaveX 充当从节点，部署从属服务 Supervisor 进程；iclient0 和 iclient1 充当客户节点，部署客户端，图 10-13 为部署规划的效果图。

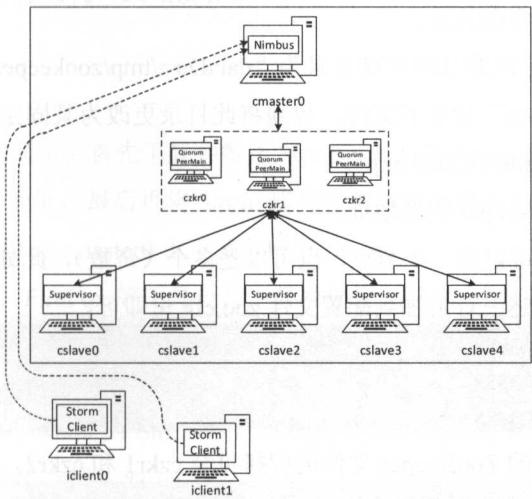


图 10-13 独立模式的 Storm 集群效果图

2) 配置机器软硬件、网络、OS 环境

制定好上述规划后, 请读者自行准备硬件机器, 各台机器的操作系统统一为 CentOS-6.7-x86_64, 除了默认的 root 用户外, 统一添加 allen 用户, 编者将以 allen 用户来安装 ZooKeeper、Storm。在 Storm 部署之前, 需要对集群中每台机器, 修改其机器名, 添加域名映射, 关闭防火墙, 安装 jdk, 这些操作命令编者已经在第 3 章讲述, 故不再赘述。

需要指明的是, 无论是 ZooKeeper 还是 Storm, 二者都不要 ssh。不过, 由于在集群环境下, 经常需要从中心机上执行 Shell 命令来操纵集群内其他或所有机器。这类操作一般都借助 ssh 完成, 读者可参考第 2 章、第 3 章、第 9 章, 独立完成 ssh 配置, 这里不再赘述。

3) 部署 ZooKeeper

由于 ZooKeeper 采用对等结构, 故在部署 ZooKeeper 时无须指定主从, 只要在配置文件里写明各 ZooKeeper 所在机地址即可, 此外, 由于 ZooKeeper 也要使用本地文件系统来存储数据, 故还要指定 ZooKeeper 的本地存储目录, 下面是在 czkr0、czkr1、czkr2 这三台机器上部署 ZooKeeper 的详细过程。

Step1 在 czkr0 上, 以 allen 身份下载并解压 ZooKeeper。

请读者使用 wget, FireFox 等工具自行下载并使用下述命令解压 ZooKeeper:

```
[allen@cmaster0 ~]$ tar -zxvf ~/zookeeper-3.4.6.tar.gz
```

Step2 将 ZooKeeper 默认配置文件 conf/zoo_sample.cfg 更名为 conf/zoo.cfg。

Step3 配置数据存储目录。

配置文件 zoo.cfg 里默认的存储目录为 “dataDir=/tmp/zookeeper”, 由于机器重启后, 系统会自动清空 “/tmp” 目录下文件, 故须将此目录更改为某固定目录, 比如将其配置为 “dataDir=/home/allen/Cloud/zkp”。

Step4 配置 ZooKeeper 集群地址。

ZooKeeper 集群可以是一个节点, 也可以是多个 (奇数), 此处配置的 ZooKeeper 集群为三个节点, 将下述三行追加到配置文件 zoo.cfg 里即可。

```
server.1=czkr0:2888:3888
server.2=czkr1:2888:3888
server.3=czkr2:2888:3888
```

Step5 将配置好的 ZooKeeper 文件远程拷贝至 czkr1 和 czkr2。

```
[allen@czkr0 ~]$ scp -r zookeeper-3.4.6 czkr1:~/
[allen@czkr0 ~]$ scp -r zookeeper-3.4.6 czkr2:~/
```

Step6 新建并填写各机 ID。

在配置的 ZooKeeper 数据存储目录 (当前为 “dataDir=/home/allen/Cloud/zkp”) 中,

新建文件 myid。三台机器按 czkr0、czkr1 和 czkr2 先后顺序分别写入“1”、“2”和“3”。

```
[allen@czkr0 ~]$ cat /home/allen/Cloud/zkp/myid          #cmaster0 机
1
[allen@czkr1 ~]$ cat /home/allen/Cloud/zkp/myid          #cmaster1 机
2
[allen@czkr2 ~]$ cat /home/allen/Cloud/zkp/myid          #cmaster2 机
3
```

Step7 确定存在数据存储目录和 myid 文件。

由于编者配置的 ZooKeeper 数据存储目录为“dataDir=/home/allen/Cloud/zkp”，所以在启动 ZooKeeper 集群之前，请读者确定 czkr0、czkr1、czkr2 这三台机已存在目录“/home/allen/Cloud/zkp”和文件“/home/allen/Cloud/zkp/myid”。

Step8 启动 ZooKeeper 集群。

使用下述命令启动 ZooKeeper：

```
[allen@czkr0 ~]$ zookeeper-3.4.6/bin/zkServer.sh start
[allen@czkr1 ~]$ zookeeper-3.4.6/bin/zkServer.sh start
[allen@czkr2 ~]$ zookeeper-3.4.6/bin/zkServer.sh start
```

显然，从 ZooKeeper 的部署和启动可以看出，ZooKeeper 采用对等结构，不过实际上，ZooKeeper 内部还是领导和被领导关系，这三个进程启动后，它们之间会自动选举出选举出一个“领导”，其他两个 ZooKeeper 则自动成为追随者，选举的原则很简单，就是少数服从多数原则，这就是为何在 ZooKeeper 集群中，节点数总是奇数的原因。

上文除了 ZooKeeper 的存储机制外，已经讲述了 ZooKeeper 主要运行机制（另一主要机制是 ZooKeeper 存储结构），此处读者无须关心选举过程，明白选举过程由 ZooKeeper 自动完成即可。

4) 部署 Storm

Storm 的部署非常简单，首先下载并解压 Storm，然后为 Storm 配置 ZooKeeper 集群地址和主服务 Nimbus 地址，最后再将 Storm 拷贝至集群中其他机器即可。

Step1 以 allen 身份在 cmaster0 上下载并解压 Storm。

请读者使用 wget，FireFox 等工具自行下载并使用下述命令解压 Storm：

```
[allen@cmaster0 ~]$ tar -zxvf ~/apache-storm-0.10.0.tar.gz
```

Step2 配置 ZooKeeper 集群地址。

conf 下的文件 storm.yaml 为 Storm 的默认配置文件，该文件中，默认情况下已经注释了 ZooKeeper 集群地址一行，如下：

```
# storm.zookeeper.servers:
#   - "server1"
#   - "server2"
```

将上述注释号去掉并将机器名改为 czkr0、czkr1、czkr2 即可，修改后，内容替换如下：

```
storm.zookeeper.servers:
```

```
- "czkr0"
- "czkr1"
- "czkr2"
```

Step3 配置 Storm 主节点地址。

在资源管理层 Storm 采用 master/slave 架构，运行 Nimbus 进程的主机称为主节点，配置 Storm 集群主节点非常简单，只要定位文件 storm.yaml 中下述一行：

```
# nimbus.host: "nimbus"
```

去掉注释，更改主机地址为 cmaster0 即可，修改后上述内容替换如下：

```
nimbus.host: "cmaster0"
```

Step4 将配置好的 Storm 文件远程拷贝至和 cslave0~cslave4, iclient0、iclient1。

One 编辑临时文件 “/home/allen/cMachines”，将下述内容写入 cMachines：

```
cslave0
cslave1
cslave2
cslave3
cslave4
iclient0
iclient1
```

Two 使用下述循环命令，将 apache-storm-0.10.0 逐个拷至 cMachines 中各机：

```
[allen@cmaster0 ~]$ pwd
```

```
/home/allen
```

```
[allen@cmaster0 ~]$ for x in `cat cMachines`;do echo $x;scp -r apache-storm-0.10.0/ allen@$x:~/;done;
```

Step5 启动 Storm。

One 运行 Storm 的前提是 ZooKeeper 集群已启动，下述命令是 ZooKeeper 的启动和关闭命令，此处以 czkr0 为例，其他两台机器（czkr1、czkr2）操作相同：

```
[allen@czkr0 zookeeper-3.4.6]$ bin/zkServer.sh start #当前已在 zookeeper-3.4.6 目录下
```

```
[allen@czkr0 zookeeper-3.4.6]$ bin/zkServer.sh stop #allen 用户,czkr0,关闭 ZooKeeper
```

```
[allen@czkr0 zookeeper-3.4.6]$ bin/zkServer.sh status #allen 用户,czkr0,查看当前状态
```

Two Storm 的启动非常简单，只需要在主节点上启动主服务，从节点上启动从服务，下面是 cmaster0 上启动主服务 Nimbus：

```
[allen@cmaster0 apache-storm-0.10.0]$ pwd #当前已在 apache-storm-0.10.0
```

```
/home/allen/apache-storm-0.10.0
```

```
[allen@cmaster0 apache-storm-0.10.0]$ bin/storm nimbus & #cmaster0,allen,启动 Nimbus
```

下面是从节点启动从服务 Supervisor：

```
[allen@cslave0 ~]$ apache-storm-0.10.0/bin/storm supervisor & #cslave0,allen,启动 Supervisor
```

```
[allen@cslave1 ~]$ apache-storm-0.10.0/bin/storm supervisor & #cslave1,allen,启动 Supervisor
```

```
[allen@cslave2 ~]$ apache-storm-0.10.0/bin/storm supervisor & #cslave2,allen,启动 Supervisor
```

```
[allen@cslave3 ~]$ apache-storm-0.10.0/bin/storm supervisor & #cslave3,allen,启动 Supervisor
```

```
[allen@cslave4 ~]$ apache-storm-0.10.0/bin/storm supervisor & #cslave4,allen,启动 Supervisor
```


符号“&”说明以后台方式运行该命令，请不要试图以 Step4 中的循环 Shell 执行上述命令。一旦开启 Storm 集群后，Storm 就会永恒运行下去，没有显示的关闭方法。对于 iclient0 和 iclient1 这两台客户机，无须开启任何 Storm 驻守服务。

Three 就像 HDFS、YARN、Spark 这类服务一样，Storm 也有 Web UI 界面，只不过 Storm 将该服务独立出来，用户需要手工启动 Web UI，才能看到该界面，其启动命令如下，该服务须和 Nimbus 服务同机。

```
[allen@cmaster0 apache-storm-0.10.0]$ bin/storm ui & #cmaster0,allen,启动 UI
```

至此 Storm 集群启动结束。

5) 验证 Storm 集群

可以采取下述方式验证 Storm 集群是否启动成功。

(1) 查看进程

One 在 cmaster0 上，使用 jps 和 ps 命令可以查看到 Nimbus 进程，UI 进程：

```
[allen@cslave1 ~]$ jps
44962 Jps
2980 core
2858 nimbus
[allen@cslave1 ~]$ ps -ef | grep storm
```

上述输出的 core 即为 UI 进程，由于 ps 命令输出太多，这里不再显示。

Two 在 cslave0~4 各机上，使用 jps 和 ps 命令都可以查看到进程 supervisor:

```
[allen@cslave1 ~]$ jps
2724 supervisor
[allen@cslave1 ~]$ ps -ef | grep supervisor
```

(2) 通过 Web UI 确定集群正常运行

在集群中任一机器的 FireFox 地址栏里，输入“http://cmaster0:8080”，回车，即可看到 Storm 集群 Web UI（图 10-14）。

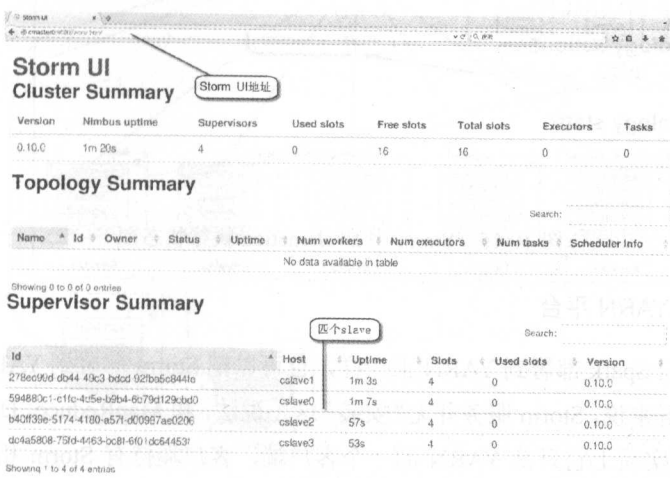


图 10-14 Storm WebUI 界面

(3) 通过在 Storm 上运行 Topology

如果能在 Storm 上成功运行 Storm-App,即可成功说明集群部署成功,关于运行 Storm 示例 Topology 的实例请参考下一示例。

6) 向 Storm 提交 Topology

①在 Storm 诸多示例程序中, ExclamationTopology 和 WordCountTopology 是两个最简单且最常被用来测试集群的 Storm-App。下面的命令就是以 ExclamationTopology 为例,实现在 iclient0 上向 Storm 集群提交该应用:

```
[allen@iclient0 ~]$ bin/storm jar examples/storm-starter/storm-starter-topologies-0.10.0.jar \  
> storm.starter.ExclamationTopology ec-0
```

在执行上述命令时,请务必确保主类后至少有一个参数(此处为 ec-0),这是因为默认情况下,当提交的 Storm-App 不带任何参数时, Storm 会以本地模式运行该 Storm-App。上述命令会根据配置文件中的 Nimbus 地址,自动向该地址提交 Storm-App。Nimbus 在收到该应用后,会分配、启动、执行该 Storm-App。

②在程序运行过程中,读者可在 iclient0 机 FireFox 地址栏里,输入“http://cmaster0:8080”,回车,即可在 Storm Web UI 界面上看到本应用。

③由于 Storm-App 是一个实时应用程序,故集群中的每个 Storm-App 都会一直运行下去,永不停止。此时若用户欲手动停止该 Storm-App,可在 Web UI 界面上选中该应用,然后点击 kill,手动停止该 Storm-App,图 10-15 为编者手动停止 ExclamationTopology 截图。



图 10-15 Storm 中 Exclamation 执行状态图

2. 宿主于 YARN 平台

就像可以把 Spark 部署到 YARN 上一样,也可以把 Storm 部署到 YARN 集群。不过对于 YARN 集群来说, Storm 服务并无“安装”这一说法,和 MapReduce、DistributedShell、Spark 等一样,实质上它只是 YARN 的一个客户端,客户端持有 Storm 相关资源(Jar、程序包、命令包等各类资源),然后向 YARN 集群提交这类资源即可。ResourceManager

会使用这些代码资源，在 YARN 集群内启动相关实体执行 Storm-App。

如图 10-16 所示，在 Storm 执行之前，YARN 集群中并未存在任何 Storm 痕迹，没有 Storm 进程，没有 Storm Jar 包，甚至 YARN 连 Storm 是谁都不知道。此时若客户端需要提交 Storm-App，那么此客户端需要持有 Storm (StormAppMaster) 和 Storm 用户层面的程序代码 (StormAppBusinessLogic)。其提交和执行过程是，首先，Client 使用相关协议向 ResourceManager 申请第一个 Container 来执行 Storm 的主服务 StormAppMaster；接着主服务 StormAppMaster 向 ResourceManager 申请一定数量的 Container 来执行 Storm 从属进程 Supervisor；这样 YARN 集群中就存在了一个 Nimbus/Supervisor 集群，该集群就是本章开始讲述的独立模式时的 Storm 集群。

当 YARN 中启动了 Storm 集群后，用户可使用 bin/storm 脚本提交 Storm-App。和独立模式一样，在用户提交 Storm-App 后，Supervisor 会启动一系列 Worker 来执行用户编写编写的 Storm 应用程序。

YARN 部署也分手工和工具两种方式，为保持 littleCstor 完整性，编者直接使用第 3 章已部署好的 YARN。

使用 Ambari 部署 Storm 可以说是一键操作，难点几乎都在 Ambari 工具本身部署上，以下步骤从无到有，简单介绍了 Ambari 自身部署和使用 Ambari 部署 Storm 的大概步骤。

Step1 制定部署规划。

Step2 准备硬件机器和 OS 环境。

Step3 配置单机 OS 环境和集群环境。

Step4 部署 Ambari-server。

Step5 使用 Ambari-server 部署 HDFS、YARN、ZooKeeper、Storm。

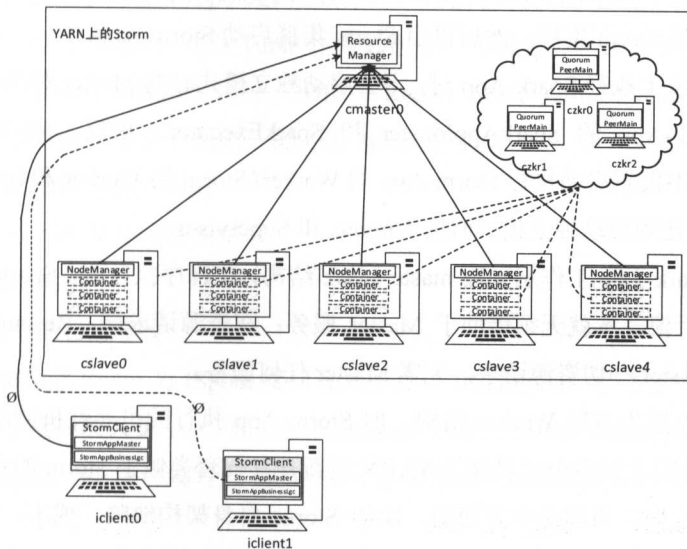


图 10-16 Spark 部署于 YARN 集群

例 2 使用 Ambari 为 littleCstor 部署 Storm。

解 由于大数据平台涉及太多组件，故部署之前最好制定一个完备的部署计划，否则极有可能导致由于各机角色分配混乱而部署失败。

表 10-1 littleCstor 上 Storm 部署规划

机 器	角 色	部署服务
cmaster1	Nimbus	Storm 资源管理主服务
	Storm UI Server	Storm UI 界面

本题以第 3 章为前提，只给出 Storm 部署规划表(表 10-1)和部署效果图(图 10-16)，具体部署过程请参见第 3 章。读者须注意，图 10-16 中 iclient0 到 cmaster0 或 cslave0、2 的链接(图中 Ø 表示)实际上并不存在，也就是 iclient0 并不需要向任何机器汇报心跳包，只有当 iclient0 需要向集群提交 storm 任务时，它才会主动连接 cmaster0。

正如图 10-16 所示，在 YARN 集群中，和 MapReduce、Pig、Hive 等组件一样，Storm 只是 YARN 的一个客户端。

例 3 简述 YARN 上 Spark-App 执行过程；简述 YARN 上 Storm-App 执行过程；试比较两者区别，并给出何原因导致这种区别。

解①当在 YARN 上执行 Spark-App 时，YARN 为 Spark 启动了 SparkAppMaster 和 SparkExecutor，然后 SparkAppMaster 指挥各 SparkExecutor 并行执行 Spark-App。

②当 Storm-App 运行在 YARN 上时，YARN 须首先利用一系列 Container 启动 Nimbus 和 Supervisor，再由 Nimbus 和 Supervisor 合作启动一系列 Worker 来执行 Storm-App。这相当于回到了独立模式时的 Storm 集群，换句话说，YARN 上运行 Storm-App 时，须先在 YARN 上启动 Storm 集群，然后再由 Storm 集群启动 Storm-App。

③在 YARN 上执行 Spark-App 时，不会启动独立模式时的 Master 和 Worker，只启动直接执行 Spark-App 的 SparkAppMaster 和 SparkExecutor。相反，当 YARN 上执行 Storm-App 时，不仅须启动执行 Storm-App 的 Worker(Storm 的 Worker 和 Spark 的 Worker 同名不同质)，还须启动独立模式时的 Nimbus 和 Supervisor。

④原因 Spark-App 工作层采用 master/slave 结构，在程序执行层，SparkAppMaster 管理程序整个执行流，本就无须借助于 Master 服务；在资源请求层，ResourceManager 接管 SparkAppMaster 一切资源请求，无需 Master 任何服务。

Storm 工作层为对等 Worker 结构，但 Storm-App 执行的可靠性机制却需要 Nimbus 维持。其实，开发人员完全可以基于 YARN 重新开发一套类似于 Storm 的实时计算框架，之所以未曾这么做，原因是多方面的，比如 Storm 自身架构缺陷、成本、必要性等。要知道，就像并非所有 Android 应用都适合 iOS 一样，对于诸多大数据组件，并不是非得把它们集成到 YARN 上才能证明它们是成功的软件。

10.1.5 计算模型

Storm-App 由多个处理节点构成的实时有向无环处理流框架，Storm 术语称此有向无环处理流为 Topology，而 Topology 中各流之间有分组的概念，特定流的处理节点有并行度概念。

1. Topology 组件

Storm 分布式计算结构称为 Topology（拓扑），由 Stream（数据流）、Spout（数据流生成者）、Bolt（运算）组成（图 10-17）。Storm Topology 大致等同于 Hadoop 的 Job，不过，Hadoop 中的 Job 有明显的启始和终止时间，Storm Topology 则会一直运行下去，永不停止。

1) Stream

Storm 的核心数据结构是 Tuple，Tuple 是包含了一个或多个键值对的列表，Stream 是一个或多个 Tuple 组成的序列。

Storm 框架底层 Tuple 为 ITuple 接口及其子接口 Tuple 和 TridentTuple，在使用时，用户可以直接使用 Tuple 接口的实现类 TupleImpl，该类可满足大部分业务需求。

2) Spout

Spout 代表了一个 Storm Topology 的数据入口，充当采集器的角色，其连接到数据源，将数据转化成一个个 Tuple，并将 Tuple 作为数据源，发射出去。目前 Spout 支持的数据源包括 Kafka、HDFS、HBase、Hive、JDBC、Redis、Solr、Azure EventHubs、Flux。总之，只要数据源符合下述基本规则，都可用 Spout 读取：

- Web 或移动程序的点击流
- Twitter 或其他社交网络的消息
- 传感器的输出
- 应用程序的日志事件

Storm 框架底层 Spout 为 ISpout 接口及其子接口 IRichSpout，为方便用户编程，框架还提供了这两个接口的大量子类，如 BaseRichSpout、DRPCSpout、KafkaSpout、ShellSpout、TwitterSampleSpout 等。用户编写的 Spout 类可直接继承 BaseRichSpout，其满足绝大部分业务需求。

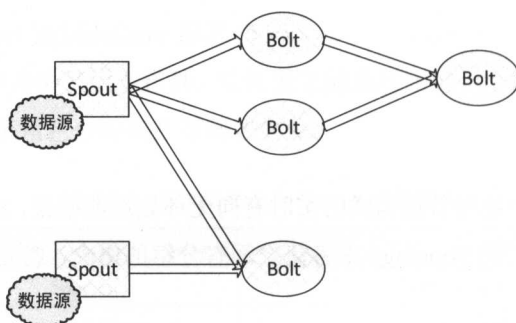


图 10-17 Topology 示例

3) Bolt

Bolt 可以理解为计算程序中的运算或者函数，将一个或者多个数据流作为输入，对数据实施运算后，选择性地输出一个或者多个数据流。**Bolt** 可以订阅多个由 **Spout** 或者其他 **Bolt** 发射的数据流，这样就可以建立复杂的数据流转换网络，**Bolt** 中常见的操作为：

- 过滤 Tuple
- 连接 (join) 和聚合操作 (aggregation)
- 计算
- 数据库读写

Storm 框架底层 Bolt 为 **IBolt** 接口及其子接口 **IRichBolt**，为方便用户编程，框架还提供了这两个接口的大量子类 **AbstractHBaseBolt**、**AbstractHdfsBolt**、**AbstractJdbcBolt**、**AbstractRedisBolt**、**BaseRichBolt** 等。用户编写的 Bolt 类可直接继承 **BaseRichBolt**，其满足绝大部分业务需求。

2. 流式实时计算模型

本节给出计算模型后，即以实例讲述该模型。

1) 计算模型理论

Storm 的计算模型是由多个处理节点构成的实时有向无环处理流。

比如现有一场景：两台生产机不断产生由纯数字或纯单词构成的 Sentence，**SentenceSpout** 则不断读取该 Sentence 并将 Word 型 Sentence 发往 **SplitWordSentenceBolt**，Number 型 Sentence 发往 **SplitNumberSentenceBolt**。这两个 **SplitXXXBolt** 再将 Sentence 拆分成一个个独立单词后，又分别发往 **WordCountBolt** 和 **NumberCountBolt**，这两个 **XXXCountBolt** 又各自完成自身任务，该处理流即是 Storm 计算模型的典型示例（图 10-18）。

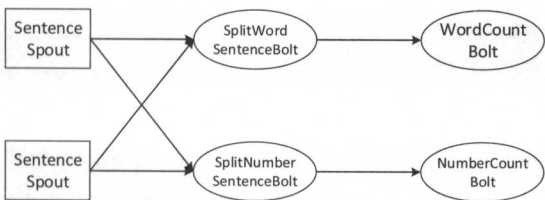


图 10-18 Storm 计算模型实例

2) 场景 1: 本地模式

场景: 请设计一实时应用程序，要求该应用能够实时统计生产机产生的单词数量。

下面给出设计方案。

(1) 常规方案

最简单的方法就是生产机上定义一个 `HashMap<String,Long>`，对于相同的 `String`，来一个则 `Long` 加一，对于不同的 `String`，来一个登记 `String` 并把 `Long` 值置一。该程序清晰明了、逻辑简单、操作性强。不过该程序的问题也很明显，比如，为确保生产机高峰时性能充足，一般不会将程序放在生产机上执行，再比如程序扩展性特别差，始终只是一个程序在干活，数据量一旦大起来，程序极有可能出现内存耗尽而崩溃或数据丢失。

(2) Storm 方案

Storm 方案就是 Storm 计算模型的实际体现，下面分思路、代码、执行步骤等逐步讲解。

Step1 思路。

当采用计算模型时，可使用 `SentenceSpout` 类读取 `Sentence` 并将 `Sentence` 发出到下一个处理节点；使用 `SplitSentenceBolt` 接收 `SentenceSpout` 发来的 `Sentence` 并将此 `Sentence` 拆分成一个个 `Word` 后，将这些 `Word` 继续发往下一个处理节点；使用 `WordCountBolt` 接收 `SplitSentenceBolt` 发来 `Word` 并进行单词统计，最后将此统计 `<Word,Long>` 继续发往下一个处理节点；使用 `ReportBolt` 类接收 `WordCountBolt` 发来的 `<Word,Long>` 并向控制汇报结果，整个程序处理流如图 10-19 所示。

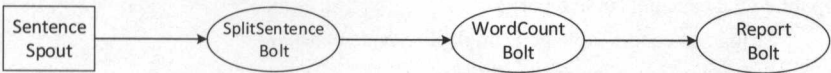


图 10-19 WCTopology

Step2 代码编写环境。

使用 Eclipse 新建 Maven 项目，在 `pom.xml` 里添加如下依赖：

```
<dependency>
  <groupId>org.apache.storm</groupId>
  <artifactId>storm-core</artifactId>
```



```
<version>0.10.0</version>
</dependency>
```

Step3 代码。

根据上诉 WCTopology 设计方案，开发如下 SentenceSpout 类独立读取生产机产生的句子。为降低复杂性，此处模拟数据源，不断发出"aa bb cc", "xx yy zz"两句：

```
public class SentenceSpout extends BaseRichSpout {
    private SpoutOutputCollector collector;
    private String[] sentences = {"aa bb cc", "xx yy zz"};
    private int index = 0;
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("sentence"));
    }
    public void open(Map config, TopologyContext context,
        SpoutOutputCollector collector) {
        this.collector = collector;
    }
    public void nextTuple() {
        this.collector.emit(new Values(sentences[index]));
        index++;
        if (index >= sentences.length) { index = 0; }
        try {TimeUnit.MICROSECONDS.sleep(100);
        } catch (InterruptedException e) {e.printStackTrace();}
    }
}
```

上述的 SentenceSpout 不断发出"aa bb cc", "xx yy zz"两个 Sentences，下面的 SplitSentenceBolt 则不停地读取 SentenceSpout 发来的所有 Sentences 并将 Sentence 拆成一个个单词，继续发出。

```
public class SplitSentenceBolt extends BaseRichBolt{
    private OutputCollector collector;
    public void prepare(Map config, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
    }
    public void execute(Tuple tuple) {
        String sentence = tuple.getStringByField("sentence");
        String[] words = sentence.split(" ");
        for(String word : words){
            this.collector.emit(new Values(word));
        }
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```


上述的 SplitSentenceBolt 将拆分后的 Word 写出后, 交由 WordCountBolt 处理, WordCountBolt 读取这些单词并将其转换为<String,Long>后, 继续发出, WordCountBolt 类代码如下:

```
public class WordCountBolt extends BaseRichBolt {
    private OutputCollector collector;
    private HashMap<String, Long> counts = null;
    public void prepare(Map config, TopologyContext context,
        OutputCollector collector) {
        this.collector = collector;
        this.counts = new HashMap<String, Long>();
    }
    public void execute(Tuple tuple) {
        String word = tuple.getStringByField("word");
        Long count = this.counts.get(word);
        if(count == null){
            count = 0L;
        }
        count++;
        this.counts.put(word, count);
        this.collector.emit(new Values(word, count));
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}
```

其实, 上述 WordCountBolt 已经处理结束, 该类里 HashMap<String,Long>的最新值即为当前实时统计单词出现次数, 由于该值实时更新, 为保障其完整性, 编者使用下述多余的 ReportBolt 类汇报结果, 须注意的是, 该类只有在本地模式下才可执行, 在集群环境下, 应去掉该类。

```
public class ReportBolt extends BaseRichBolt {
    private HashMap<String, Long> counts = null;
    public void prepare(Map config, TopologyContext context, OutputCollector collector) {
        this.counts = new HashMap<String, Long>();
    }
    public void execute(Tuple tuple) {
        String word = tuple.getStringByField("word");
        Long count = tuple.getLongByField("count");
        this.counts.put(word, count);
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        // this bolt does not emit anything
    }
}
```

```

@Override
public void cleanup() {
    System.out.println("--- FINAL COUNTS ---");
    List<String> keys = new ArrayList<String>();
    keys.addAll(this.counts.keySet());
    Collections.sort(keys);
    for (String key : keys) {
        System.out.println(key + " : " + this.counts.get(key));
    }
    System.out.println("-----");
}
}

```

为观察 Storm-App 执行过程，编者为 WCTopology 添加了 ReportBolt 类，该类在 Storm-App 结束时，会汇报 HashMap<String,Long>里的最终结果。下面的 main 方法用来将这几个处理节点组成一个 Topology，接着以本地方式，提交该拓扑：

```

public class WordCountTopology {
    public static void main(String[] args) throws Exception {
        SentenceSpout spout = new SentenceSpout();
        SplitSentenceBolt splitBolt = new SplitSentenceBolt();
        WordCountBolt countBolt = new WordCountBolt();
        ReportBolt reportBolt = new ReportBolt();
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("sentence-spout", spout, 2).setNumTasks(1);
        // SentenceSpout --> SplitSentenceBolt
        builder.setBolt("split-bolt", splitBolt).shuffleGrouping("sentence-spout");
        // SplitSentenceBolt --> WordCountBolt
        builder.setBolt("count-bolt", countBolt).fieldsGrouping("split-bolt", new Fields("word"));
        // WordCountBolt --> ReportBolt
        builder.setBolt("report-bolt", reportBolt).globalGrouping("count-bolt");
        Config config = new Config();
        LocalCluster cluster = new LocalCluster();
        cluster.submitTopology("word-count-topology", config, builder.createTopology());
        waitForSeconds(6);
        cluster.killTopology("word-count-topology");
        cluster.shutdown();
    }
}

```

Step4 编译执行。

程序开发结束，为确保项目已编译，不妨左键选中该 Maven 项目，右键单击出菜单项，依次进入 Run As→Maven test。最后，左键选中类 WordCountTopology，右键单击出菜单，依次进入 Run As→Java Application。程序执行 6 秒后，将自动退出并输出如下结果：

```
--- FINAL COUNTS ---
```

```

aa : 901
bb : 901
cc : 901
xx : 900
yy : 900
zz : 900
-----

```

请读者务必注意，在 `WordCountTopology` 主方法里，编者已写定本 `Topology` 使用本地模式，故无法在集群上执行该应用；此外由于 `SentenceSpout` 里模拟数据生成的代码执行过快，已大大超出后续处理节点处理频率，故请务必不要删除 `SentenceSpout` 里线程休眠代码块；最后，在 `WordCountTopology` 类里，编者指定在程序执行 6 秒后，自动退出，请读者调试此处代码时，确保时间大于等于 6，因为时间过短会直接导致程序无法执行而无任何输出。

`Storm` 集群中运行的 `Storm-App` 会一直运行下去，除非手动停止，否则永不停止。`Spout` 永不停止地读取数据源生成的数据并实时将数据发往各 `Bolt`，各个 `Bolt` 永不停止地接收上一层 `Spout` 或 `Bolt` 发来的 `Tuple` 并在处理完后继续发往下一层 `Bolt`。站在某特定 `Tuple` 的角度上，假定该 `Tuple` 经过：`Spout`→`BoltA`→`BoltB`→...→`BoltX`，从它进入 `Spout`，出 `Spout`，进入 `BoltA`，出 `BoltA`，直到出 `BoltX`，其 (`Tuple`) 生命周期已经结束，可是 `Storm-App` 永不停止，有 `Tuple` 过来，则处理，无 `Tuple`，则空操作，永不停歇，这是实时计算的精髓。

3) 场景 2: 集群模式

场景：请修改“场景 1”代码，使其能够在 `Storm` 集群上执行。

Step1 修改代码。

首先，当在集群中运行 `WordCountTopology` 时，`ReportBolt` 的 `cleanup` 方法将不会执行，故在下面主类中，不应再使用该类。其次，当向真实集群提交 `Storm-App` 时，须将代码中的本地模式更改为集群模式，具体代码如下：

```

public class WordCountTopology {
    public static void main(String[] args) throws Exception {
        SentenceSpout spout = new SentenceSpout();
        SplitSentenceBolt splitBolt = new SplitSentenceBolt();
        WordCountBolt countBolt = new WordCountBolt();
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("sentence-spout", spout);
        // SentenceSpout --> SplitSentenceBolt
        builder.setBolt("split-bolt", splitBolt).shuffleGrouping("sentence-spout");
        // SplitSentenceBolt --> WordCountBolt
        builder.setBolt("count-bolt", countBolt).fieldsGrouping("split-bolt", new Fields("word"));
    }
}

```

```
Config conf = new Config();
conf.setNumWorkers(3);
StormSubmitter.submitTopologyWithProgressBar(args[0], conf, builder.createTopology());
}
```

除了 WordCountTopology 外, 其他三个类 (即 SentenceSpout、SplitSentenceBolt 和 WordCountBolt) 无须更改。

Step2 编译打包。

程序开发结束后, 可使用 Maven 来打包该项目。左键选中该 Maven 项目, 右键单击出菜单项, 依次进入 Run As→Maven build, 在弹出对话框里的 Goals 写入 Package, 单击此对话框的 Apply, Run 后, Maven 会自动打包该项目。

进入项目 src 同级 target 目录, 找到打好包的文件, 比如编者的为 storm.example-0.0.1.jar。

最后将此 Jar 包上传至 iclient0, 在 iclient0 上, 进入 Storm 客户端, 使用下述命令向 Nimbus 提交本应用:

```
[allen@iclient0 apache-storm-0.10.0]$ pwd
/home/allen/apache-storm-0.10.0
[allen@iclient0 apache-storm-0.10.0]$ bin/storm jar ~/storm.example-0.0.1.jar \
> bg.storm.example.WordCountTopology wc-0
```

Step3 终止应用。

一旦向集群提交 Storm-App, 其会一直运行下去, 永不停止。在程序执行过程中, 其不但会占用大量内存资源, 还会不停刷新日志至本地磁盘, 时间一久 (一周以上), 磁盘将会耗尽, 对于正在开发的 Storm-App, 建议观察一段时间后, 手工停止该应用。

3. 节点并行度

Storm-App 节点并行度指的是启动多个 Worker 或 Executor 来执行 Topology 中的 Spout/Bolt。以 WordCountTopology 为例, 假定其 Topology 为 SentenceSpout→SplitSentenceBolt→WordCountBolt, 则 SentenceSpout、SplitSentenceBolt、WordCountBolt 三者之间逐个依赖, 不好并行处理。不过, 可以启动多个 SentenceSpout, 此时这些 SentenceSpout 之间都是并行执行的。同理可设置多个 SplitSentence, 多个 WordCountBolt。

当设置了 Topology 中某 Spout/Bolt 并行度时, 实际上这个 Spout/Bolt 就在并行执行, 这就是所谓的 Task 并行。

1) 局部并行

Topology 由各个 Spout、Bolt 实例组成, 局部并行指的是设置某 Spout/Bolt 并行度。比如由 SentenceSpout→SplitSentenceBolt→WordCountBolt 构成的 WCTopology, 可以单独设置 SentenceSpout 的并行度, 可以单独设置 SplitSentence 的并行度, 也可以单独设置

WordCountBolt 的并行度。

(1) 线程级并行

默认, Storm-App 都会在一个 Worker 上执行, 此时若设置了某 Spout/Bolt 并行度, 则 Worker 会启动相应数量 Executor 线程执行该设置。例如如下代码就是要求 Nimbus 为此 WordCountTopology 分配两个线程执行两个 SentenceSpout 任务, 即一个线程完成一个 SentenceSpout 任务:

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("sentence-spout", spout, 2).setNumTasks(2);
//第一个“2”为设置 Executor 线程数量, 第二个“2”为设置 SentenceSpout 任务数量
```

虽然这两个线程都在 Worker 进程内, 不过由于线程间并发执行, 故这两个 SentenceSpout 任务就在并行执行 (单机 CPU 多核), 其执行步骤如图 10-20 所示。

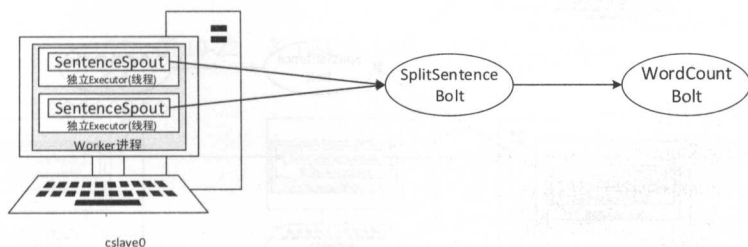


图 10-20 线程间并发

(2) 进程级并行

当在 Storm 集群上运行 Storm-App 时, 可以设置该 Storm-App 持有多个 Worker 进程, Nimbus 会尽量将 Storm-App 的各个 Task (Spout/Bolt 实例) 均匀分散于各 Worker。比如如下代码要求 Nimbus 为此 WordCountTopology 分配两个线程执行两个 SentenceSpout 任务, 即一个线程完成一个 SentenceSpout 任务; 同时还申请了两个 Worker 来执行该 WCTopology, 故在实际调度时, 两个进程上分别启动一个 Executor 线程来执行 SentenceSpout:

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("sentence-spout", spout, 2).setNumTasks(2);
//第一个“2”为设置 Executor 线程数量, 第二个“2”为设置 SentenceSpout 任务数量
Config conf = new Config();
conf.setNumWorkers(2); //本 Storm-App 持有两个 Worker 进程
StormSubmitter.submitTopologyWithProgressBar("s", conf, builder.createTopology());
```

显然, 这两个 Executor 线程运行于不同机器不同进程上, 完全独立。cslave0 上的 SentenceSpout 和 cslave1 上的 SentenceSpout 也只是属于同一数据结构 (类), 无任何其他关系, 故无论从哪个角度上来说, 这两个 SentenceSpout 就是在并行执行 (图 10-21)。

（3）混合并行

顾名思义，该方式就是同时采用线程级并发和进程级并行，须注意的是，该模式下 Topology 中的 Task 数须大于 Worker 数。比如如下申明了该 Topology 须执行 4 个 SentenceSpout 任务，接着又申请了 4 个线程来执行这 4 个 SentenceSpout，可集群中只有两个 Worker，故按正常逻辑是每个 Worker 上开启两个线程，每个线程独立完成一个 SentenceSpout，事实上也是如此分配的，系统依旧是将所有 Task（Spout/Bolt 实例）均匀分散到各个 Worker 上。

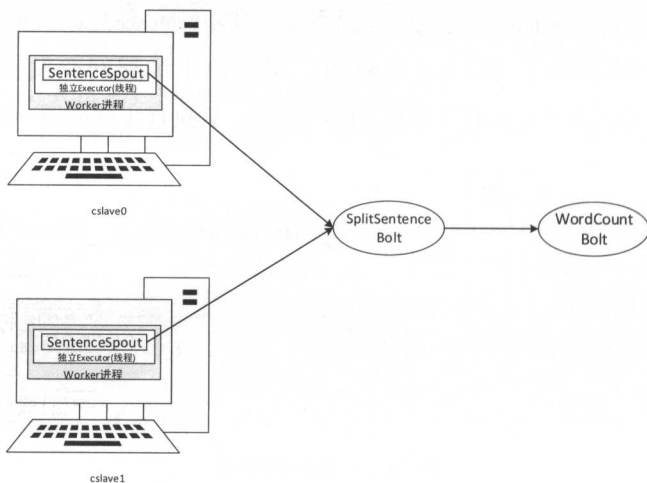


图 10-21 进程级并行

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("sentence-spout", spout, 4).setNumTasks(4);
//第一个“4”为设置 Executor 线程数量，第二个“4”为设置 SentenceSpout 任务数量
Config conf = new Config();
conf.setNumWorkers(2);
//本 Storm-App 持有两个 Worker 进程
StormSubmitter.submitTopologyWithProgressBar("s", conf, builder.createTopology());
```

显然，这两个机器上 Worker 进程无任何联系，Worker 进程内的两个线程也是并发（cslave0、cslave1 都需多核 CPU）执行，图 10-22 为执行示意图。

2）整体并行

当 Topology 的各个 Spout、Bolt 实例都设置了并行度时，该 Topology 的逐个运行节点间实质上为流水线作业，虽不是严格并行，但整体上，趋向于并行。

比如某由 SentenceSpout→SplitSentenceBolt→WordCountBolt 构成的 WCTopology（图 10-23），若 SentenceSpout 的并行度设置为 4，SplitSentence 的并行度设置为 3，则该 WordCount 拓扑趋于流水线并行。

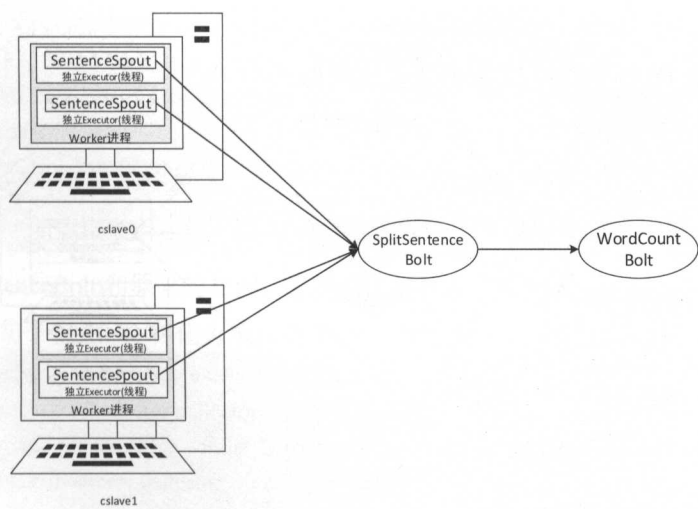


图 10-22 多 Task 多 Worker 时并行示意图

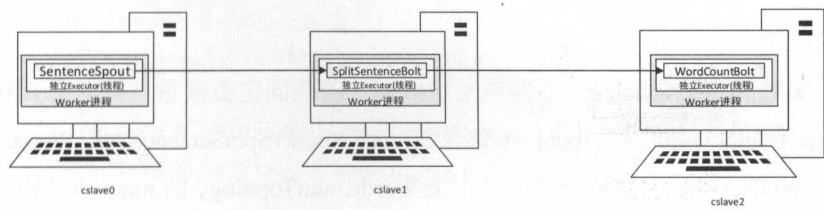


图 10-23 WordCountTopology 示例图

显然，运行于 cslave0 与 cslave1 上的四个 SentenceSpout 任务之间是并行的，运行于 cslave2、3、4 上的三个 SplitSentence 也是并行的。

当 WCTopology 各个执行点并行度都为 1 时，各个执行点间逐个依赖（图 10-23），而当 WCTopology 中有多个执行点并行度大于 1 时，显然，各个执行点间已不再逐个依赖，趋向于超线性加速比趋势。正如图 10-24 所示，本来，原本 SplitSentenceBolt 依赖于 SentenceSpout，可当 SentenceSpout、SplitSentenceBolt 都设置了并行度时，SentenceSpout 和 SplitSentenceBolt 之间呈现流水线加速。

4. 调试 Storm-App

调试 Topology 对于理解 Storm 编程模型非常重要，下面给出一场景及其各种调试方法，请读者参照该场景分析理解 Storm 编程模型。

场景：现有一个“SentenceSpout→SplitSentenceBolt→WordCountBolt→ReportBolt”结构的 WordCountTopology，给出此 Topology 各种调试方法。

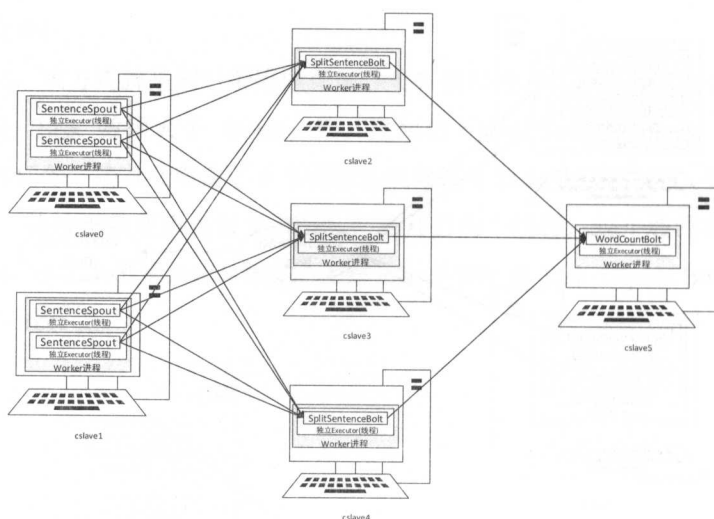


图 10-24 流水线

1) 前提: 固定数据量

由于实时的 WCTopology 不停地发送数据, 若能固定数据量, 则非常有利于调试 Storm-App。下面的 WordCountTopology 包含 SentenceSpout、SplitSentenceBolt、WordCountBolt 和 ReportBolt 四个类, 且这四个类通过主类 WordCountTopology 的 main 方法构成一本地 Storm-App, 下面为这四个类完整代码。

类 SentenceSpout 用于采集原始数据 (这里模拟了一个数据源), 请注意, 编者声明了一个本地线程 ThreadLocal, 并在 next() 方法中控制了每个 Thread 只能发出两个句子。

```
public class SentenceSpout extends BaseRichSpout {
    private SpoutOutputCollector collector;
    private String[] sentences = { "aa bb cc", "xx yy zz" };
    private int index = 0;
    private static ThreadLocal<Integer> local = new ThreadLocal<Integer>() {
        public Integer initialValue() {
            return 0;
        }
    };
};

public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("sentence"));
}

public void open(Map config, TopologyContext context, SpoutOutputCollector collector) {
    this.collector = collector;
}

public void nextTuple() {
    if (local.get() >= 2) {return;}
    this.collector.emit(new Values(sentences[index]));
}
```

```

        System.out.println("AAA "+Thread.currentThread().getName()+" "+this+" "+sentences[index]);
        index++;
        if (index >= sentences.length) {index = 0;}
        local.set(local.get() + 1);
        try {TimeUnit.MICROSECONDS.sleep(100);} catch (InterruptedException e) {e.printStackTrace();}
    }
}

```

SplitSentenceBolt 和原来的相同，无须做任何更改，其主要功能是接收发送来的句子型 Tuple，将句子型 Tuple 转化为单词型 Tuple，继续发出，代码如下：

```

public class SplitSentenceBolt extends BaseRichBolt {
    private OutputCollector collector;
    public void prepare(Map config, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
    }
    public void execute(Tuple tuple) {
        String sentence = tuple.getStringByField("sentence");
        String[] words = sentence.split(" ");
        for(String word : words){this.collector.emit(new Values(word));}
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}

```

类 WordCountBolt 用于实时统计单词个数，其代码如下：

```

public class WordCountBolt extends BaseRichBolt {
    private OutputCollector collector;
    private HashMap<String, Long> counts = null;
    public void prepare(Map config, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
        this.counts = new HashMap<String, Long>();
    }
    public void execute(Tuple tuple) {
        String word = tuple.getStringByField("word");
        Long count = this.counts.get(word);
        if(count == null){ count = 0L; }
        count++;
        this.counts.put(word, count);
        this.collector.emit(new Values(word, count));
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}

```

类 `ReportBolt` 用于观测最终结果，在 `cleanup()` 方法里，编者写明了程序结束时打印输出语句，请注意该方法只有本地测试时有效，在集群上执行时，该方法失效：

```
public class ReportBolt extends BaseRichBolt {
    private HashMap<String, Long> counts = null;
    public void prepare(Map config, TopologyContext context, OutputCollector collector) {
        this.counts = new HashMap<String, Long>();
    }
    public void execute(Tuple tuple) {
        String word = tuple.getStringByField("word");
        Long count = tuple.getLongByField("count");
        this.counts.put(word, count);
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        // this bolt does not emit anything
    }
    @Override
    public void cleanup() {
        System.out.println("--- Final Counts ---");
        List<String> keys = new ArrayList<String>();
        keys.addAll(this.counts.keySet());
        Collections.sort(keys);
        for (String key : keys) {
            System.out.println(key + " : " + this.counts.get(key));
        }
        System.out.println("----- App End -----");
    }
}
```

类 `WordCountTopology` 用来定义整个 `Topology`。

```
public class WordCountTopology {
    public static void main(String[] args) throws Exception {
        SentenceSpout spout = new SentenceSpout();
        SplitSentenceBolt splitBolt = new SplitSentenceBolt();
        WordCountBolt countBolt = new WordCountBolt();
        ReportBolt reportBolt = new ReportBolt();
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("sentence-spout", spout, 1).setNumTasks(1);
        //builder.setSpout("sentence-spout", spout, 1).setNumTasks(2);
        //builder.setSpout("sentence-spout", spout, 2).setNumTasks(2);
        //builder.setSpout("sentence-spout", spout, 2).setNumTasks(4);
        //builder.setSpout("sentence-spout", spout, 4).setNumTasks(4);
        // SentenceSpout --> SplitSentenceBolt
        builder.setBolt("split-bolt", splitBolt, 1).setNumTasks(1).shuffleGrouping("sentence-spout");
        //等于 builder.setBolt("split-bolt", splitBolt).shuffleGrouping("sentence-spout");
    }
}
```

```

// SplitSentenceBolt --> WordCountBolt
builder.setBolt("count-bolt", countBolt).fieldsGrouping("split-bolt", new Fields("word"));
// WordCountBolt --> ReportBolt
builder.setBolt("report-bolt", reportBolt).globalGrouping("count-bolt");
Config config = new Config();
LocalCluster cluster = new LocalCluster();
cluster.submitTopology("word-count-topology", config, builder.createTopology());
waitForSeconds(8);
cluster.killTopology("word-count-topology");
cluster.shutdown();
}
}

```

程序开发结束，为确保项目已编译，请左键选中该 Maven 项目，右键单击出菜单项，依次进入 Run As→Maven test。接着，左键选中类 WordCountTopology，右键单击出菜单，依次进入 Run As→Java Application。程序执行 8 秒后，将自动退出并输出结果。

2) 1 个 Task、1 个 Executor

当注释下三句，启动 builder.setSpout("sentence-spout", spout,1).setNumTasks(1)一句：

```

builder.setSpout("sentence-spout", spout,1).setNumTasks(1);
//builder.setSpout("sentence-spout", spout,1).setNumTasks(2);
//builder.setSpout("sentence-spout", spout,2).setNumTasks(2);
//builder.setSpout("sentence-spout", spout,2).setNumTasks(4);
//builder.setSpout("sentence-spout", spout,4).setNumTasks(4);

```

运行程序，可得如下结果：

```

--- Final Counts ---
aa : 1
bb : 1
cc : 1
xx : 1
yy : 1
zz : 1
----- App End -----

```

该句指示 WCTopology 申请一个（前面的“1”）Executor 来执行一个（后面的“1”）SentenceSpout，即 WCTopology 只拥有一个 Executor，且此 Executor 只需要执行一个 SentenceSpout（图 10-25）。

此时 WCTopology 只有一个 Executor 线程，且其要执行一个 SentenceSpout 对象，而该线程受到 if (local.get() >= 2) {return;} 控制，故当第一次执行 next() 方法时，其发出：

```
"aa bb cc"
```

第二次执行 next() 方法时，发出：

```
"xx yy zz"
```


从第三次执行 next()方法开始，直接 return，不再发出任何数据。

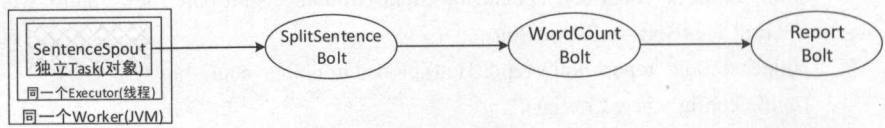


图 10-25 1 个 Task、1 个 Executor

综上，next 共计发出两个 Sentence ("aa bb cc"、"xx yy zz")，故最终结果为：1 个 aa，1 个 bb，1 个 cc，1 个 xx，1 个 yy，1 个 zz。

3) 2 个 Task、1 个 Executor

当启用 builder.setSpout("sentence-spout", spout,1).setNumTasks(2)一句时：

```
//builder.setSpout("sentence-spout", spout,1).setNumTasks(1);
builder.setSpout("sentence-spout", spout,1).setNumTasks(2);
//builder.setSpout("sentence-spout", spout,2).setNumTasks(2);
//builder.setSpout("sentence-spout", spout,2).setNumTasks(4);
//builder.setSpout("sentence-spout", spout,4).setNumTasks(4);
```

运行程序，执行结果最有可能如下：

```
--- Final Counts ---
aa : 2
bb : 2
cc : 2
----- App End -----
```

该句指示 WCTopology 申请一个（前面的“1”）Executor 来执行两个（后面的“2”）SentenceSpout，即 WCTopology 只拥有一个 Executor，且此一个 Executor 须执行两个 SentenceSpout 对象（图 26）。

此时 WCTopology 只有一个 Executor 线程，且其要执行两个 SentenceSpout 对象，而该线程受到 if(local.get() >= 2) {return;}控制，故当第一次执行 next()方法时，线程作用于第一个 SentenceSpout 对象，此时第一个 SentenceSpout 发出：

```
"aa bb cc"
```

第二次执行 next()方法时，按线程调度策略，该线程需作用于第二个 SentenceSpout 对象，第二个 SentenceSpout 也发出：

```
"aa bb cc"
```

从第三次执行 next()方法开始，直接 return，不再发出任何数据（图 10-26）。

综上，next 共计发出两个 Sentence ("aa bb cc"、"aa bb cc")，故最终结果为：2 个 aa，2 个 bb，2 个 cc。

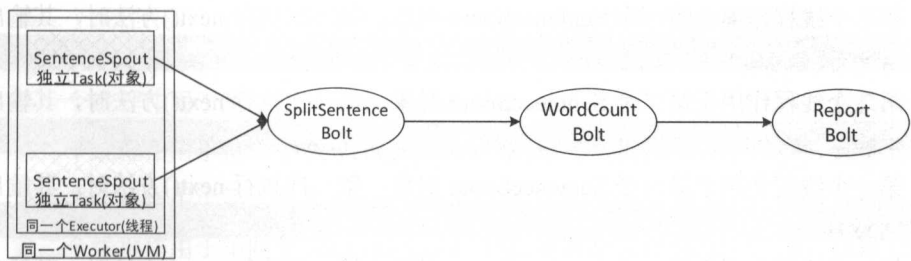


图 10-26 2 个 Task、1 个 Executor

4) 2 个 Task、2 个 Executor

当启用 `builder.setSpout("sentence-spout", spout,2).setNumTasks(2)` 一句时：

```
//builder.setSpout("sentence-spout", spout,1).setNumTasks(1);  
//builder.setSpout("sentence-spout", spout,1).setNumTasks(2);  
builder.setSpout("sentence-spout", spout,2).setNumTasks(2);  
//builder.setSpout("sentence-spout", spout,2).setNumTasks(4);  
//builder.setSpout("sentence-spout", spout,4).setNumTasks(4);
```

运行程序，执行结果最有可能如下：

```
--- Final Counts ---  
aa : 2  
bb : 2  
cc : 2  
xx : 2  
yy : 2  
zz : 2  
----- App End -----
```

该句指示 WCTopology 申请两个（前面的“2”）Executor 来执行两个（后面的“2”）SentenceSpout，即 WCTopology 拥有两个 Executor，且此两个 Executor 须执行两个 SentenceSpout 对象。故分配时实际上是一个 Executor 执行一个 SentenceSpout（图 10-27）。

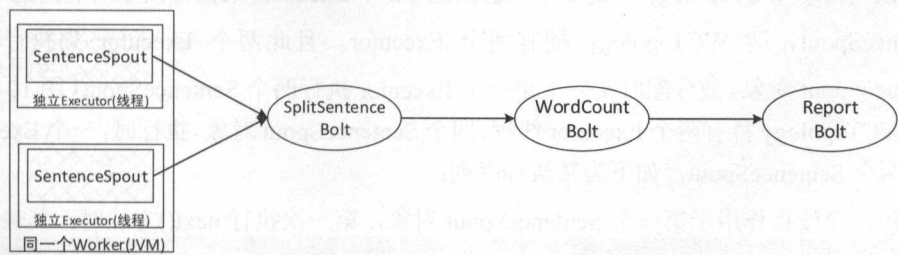


图 10-27 2 个 Task、2 个 Executor

此时 WCTopology 持有两个 Executor 线程、两个 SentenceSpout。执行时，一个 Executor 执行一个 SentenceSpout。某执行序列如下：

第一个线程作用于第一个 SentenceSpout 对象，第一次执行 next()方法时，其输出：

```
"aa bb cc"
```

第二个线程作用于第二个 SentenceSpout 对象，第一次执行 next()方法时，其输出：

```
"aa bb cc"
```

第一个线程作用于第一个 SentenceSpout 对象，第二次执行 next()方法时，其输出：

```
"xx yy zz"
```

第二个线程作用于第二个 SentenceSpout 对象，第二次执行 next()方法时，其输出：

```
"xx yy zz"
```

第一个线程作用于第一个 SentenceSpout 对象，第三次执行 next()方法时，直接 return。

第二个线程作用于第二个 SentenceSpout 对象，第三次执行 next()方法时，直接 return。

后续过程中两个线程交替执行，不过全部都不做输出。

综上，next 共计发出两个 Sentence ("aa bb cc"、"xx yy zz"、"aa bb cc"、"xx yy zz")，

故最终结果为：2 个 aa，2 个 bb，2 个 cc，2 个 xx，2 个 yy，2 个 zz。

5) 4 个 Task、2 个 Executor

当启用 builder.setSpout("sentence-spout", spout,2).setNumTasks(4)一句时：

```
//builder.setSpout("sentence-spout", spout,1).setNumTasks(1);
//builder.setSpout("sentence-spout", spout,1).setNumTasks(2);
//builder.setSpout("sentence-spout", spout,2).setNumTasks(2);
builder.setSpout("sentence-spout", spout,2).setNumTasks(4);
//builder.setSpout("sentence-spout", spout,4).setNumTasks(4);
```

运行程序，执行结果最有可能如下：

```
--- Final Counts ---
aa : 4
bb : 4
cc : 4
----- App End -----
```

该句指示 WCTopology 申请 2 个（前面的“2”）Executor 来执行四个（后面的“4”）SentenceSpout，即 WCTopology 拥有两个 Executor，且此两个 Executor 须执行四个 SentenceSpout 对象。故分配时实际上是一个 Executor 执行两个 SentenceSpout（图 10-28）。

WCTopology 持有两个 Executor 线程、四个 SentenceSpout 对象。执行时，一个 Executor 执行两个 SentenceSpout，如下为某执行序列：

第一个线程作用于第一个 SentenceSpout 对象，第一次执行 next()方法时，其输出：

```
"aa bb cc"
```

第二个线程作用于第二个 SentenceSpout 对象，第一次执行 next()方法时，其输出：

```
"aa bb cc"
```

第一个线程作用于第三个 SentenceSpout 对象，第二次执行 next()方法时，其输出：

```
"aa bb cc"
```


第二个线程作用于第四个 SentenceSpout 对象，第二次执行 next() 方法时，其输出：

"aa bb cc"

第一个线程作用于第一个 SentenceSpout 对象，第三次执行 next() 方法时，直接 return。

第二个线程作用于第二个 SentenceSpout 对象，第三次执行 next() 方法时，直接 return。

第一个线程作用于第三个 SentenceSpout 对象，第四次执行 next() 方法时，直接 return。

第二个线程作用于第四个 SentenceSpout 对象，第四次执行 next() 方法时，直接 return。

后续过程中两个线程交替执行，不过全部都不做输出。

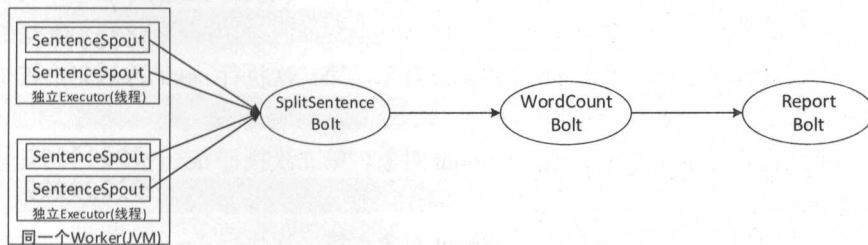


图 10-28 4 个 Task、2 个 Executor

综上，next 共计发出两个 Sentence ("aa bb cc"、"aa bb cc"、"aa bb cc"、"aa bb cc")，故最终结果为：4 个 aa，4 个 bb，4 个 cc。

6) 4 个 Task、4 个 Executor

当启用 builder.setSpout("sentence-spout", spout,4).setNumTasks(4) 一句时：

```
//builder.setSpout("sentence-spout", spout,1).setNumTasks(1);
//builder.setSpout("sentence-spout", spout,1).setNumTasks(2);
//builder.setSpout("sentence-spout", spout,2).setNumTasks(2);
//builder.setSpout("sentence-spout", spout,2).setNumTasks(4);
builder.setSpout("sentence-spout", spout,4).setNumTasks(4);
```

运行程序，执行结果最有可能如下：

```
--- Final Counts ---
aa : 4
bb : 4
cc : 4
xx : 4
yy : 4
zz : 4
----- App End -----
```

该句指示 WCTopology 申请 4 个（前面的“4”）Executor 来执行 4 个（后面的“4”）SentenceSpout，即 WCTopology 拥有 4 个 Executor，且此 4 个 Executor 须执行 4 个 SentenceSpout 对象。故分配时实际上是一个 Executor 执行 1 个 SentenceSpout（图 10-29）。

WCTopology 持有 4 个 Executor 线程、4 个 SentenceSpout 任务。执行时，一个 Executor

执行一个 SentenceSpout，如下为某执行序列：

第一个线程作用于第一个 SentenceSpout 对象，第一次执行 next()方法时，其输出：

"aa bb cc"

第二个线程作用于第二个 SentenceSpout 对象，第一次执行 next()方法时，其输出：

"aa bb cc"

第三个线程作用于第三个 SentenceSpout 对象，第一次执行 next()方法时，其输出：

"aa bb cc"

第四个线程作用于第四个 SentenceSpout 对象，第一次执行 next()方法时，其输出：

"aa bb cc"

第一个线程作用于第一个 SentenceSpout 对象，第二次执行 next()方法时，其输出：

"xx yy zz"

第二个线程作用于第二个 SentenceSpout 对象，第二次执行 next()方法时，其输出：

"xx yy zz"

第三个线程作用于第三个 SentenceSpout 对象，第二次执行 next()方法时，其输出：

"xx yy zz"

第四个线程作用于第四个 SentenceSpout 对象，第二次执行 next()方法时，其输出：

"xx yy zz"

第一个线程作用于第一个 SentenceSpout 对象，第三次执行 next()方法时，直接 return。

第二个线程作用于第二个 SentenceSpout 对象，第三次执行 next()方法时，直接 return。

第三个线程作用于第三个 SentenceSpout 对象，第三次执行 next()方法时，直接 return。

第四个线程作用于第四个 SentenceSpout 对象，第三次执行 next()方法时，直接 return。

后续过程中四个线程交替执行，不过全部都不做输出。

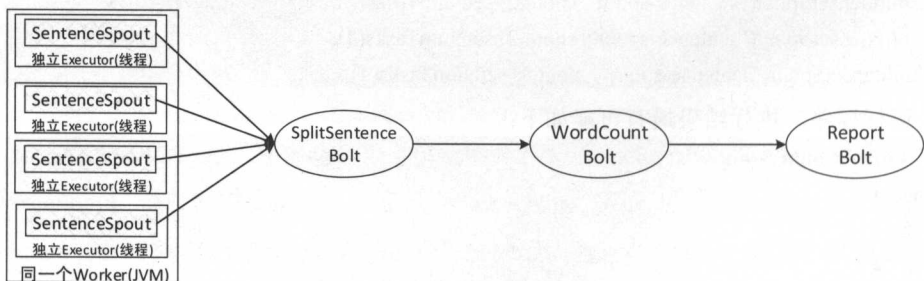


图 10-29 4 个 Task、4 个 Executor

综上，next 共计发出两个 Sentence ("aa bb cc"、"aa bb cc"、"aa bb cc"、"aa bb cc"、"xx yy zz"、"xx yy zz"、"xx yy zz"、"xx yy zz")，故最终结果为：4 个 aa，4 个 bb，4 个 cc，4 个 xx，4 个 yy，4 个 zz。

SplitSentenceBolt，WordCountBolt 并行度的设置方式和 SentenceSpout 类似，请读者自行完成，不过，不可设置 ReportBolt 并行度，这是因为 ReportBolt 在流分组时采用

globalGrouping，在此种机制下，整个 WCTopology 内，只可以有一个 ReportBolt 任务。若将上述任务提交到 Storm 集群执行，则要去掉 ReportBolt 类，且对主类做出修改。

7) 集群环境

若将上述任务提交到 Storm 集群执行，除了需要去掉 ReportBolt 类外，还要对主类做如下修改：

```
public class WordCountTopology {
    public static void main(String[] args) throws Exception {
        SentenceSpout spout = new SentenceSpout();
        SplitSentenceBolt splitBolt = new SplitSentenceBolt();
        WordCountBolt countBolt = new WordCountBolt();
        ReportBolt reportBolt = new ReportBolt();
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("sentence-spout", spout, 1).setNumTasks(1)
        //builder.setSpout("sentence-spout", spout, 1).setNumTasks(2);
        //builder.setSpout("sentence-spout", spout, 2).setNumTasks(2);
        //builder.setSpout("sentence-spout", spout, 2).setNumTasks(4);
        //builder.setSpout("sentence-spout", spout, 4).setNumTasks(4);
        // SentenceSpout --> SplitSentenceBolt
        builder.setBolt("split-bolt", splitBolt, 1).setNumTasks(1).shuffleGrouping("sentence-spout");
        //等于 builder.setBolt("split-bolt", splitBolt).shuffleGrouping("sentence-spout");
        // SplitSentenceBolt --> WordCountBolt
        builder.setBolt("count-bolt", countBolt).fieldsGrouping("split-bolt", new Fields("word"));
        Config conf = new Config();
        conf.setNumWorkers(1);
        StormSubmitter.submitTopologyWithProgressBar(args[0], conf, builder.createTopology());
    }
}
```

接着打包该 Maven 项目：左键选中该 Maven 项目，右键单击出菜单项，依次进入 Run As→Maven build，在弹出对话框里的 Goals 写入 Package，单击此对话框的 Apply，Run 后，Maven 会自动打包该项目。

进入项目 src 同级 target 目录，找到打好包的文件，比如编者的为 storm.example-0.0.1.jar。

最后将此 Jar 包上传至 iclient0，在 iclient0 上，进入 Storm 客户端，使用下述命令向 Nimbus 提交本应用：

```
[allen@iclient0 apache-storm-0.10.0]$ pwd
/home/allen/apache-storm-0.10.0
[allen@iclient0 apache-storm-0.10.0]$ bin/storm jar ~/storm.example-0.0.1.jar \
> bg.storm.example.WordCountTopology wc-0
```

请读者逐个调试 SentenceSpout 并行度的那几行代码，到集群上观察 WCTopology 并

行情况。

8) 设置 Worker 并行度

在上述 WCTopology 执行过程中，默认只有一个 Worker 进程，Storm-App 支持设置多个 Worker。若某 Storm-App 设置了多个 Worker，则 Nimbus 会将此 Storm-App 的所有 Task 尽量均匀分散到各个 Worker 上。

下面依旧使用上述场景代码，调试分析 Worker 并行度。理论上，可以为 SentenceSpout、SplitSentenceBolt、WordCountBolt 设置并行度，为简单起见，下面仅以 SentenceSpout 为例，下述代码中，builder.setSpout("sentence-spout", spout,4).setNumTasks(4) 一句表明本 Topology 申请了 4 个 Executor 线程来执行 4 个 SentenceSpout，由于 WCTopology 只有两个 Worker，故这 4 个 Executor 会均匀分散于这两个 Worker。

```
builder.setSpout("sentence-spout", spout,4).setNumTasks(4);
Config conf = new Config();
conf.setNumWorkers(2);
```

执行时，由于本 Topology 只持有两个 Worker，实际上 SplitSentenceBolt、WordCountBolt 都是在这两个 Worker 里执行的，不过在图 10-30 中，并未给出 SplitSentenceBolt 执行状态。

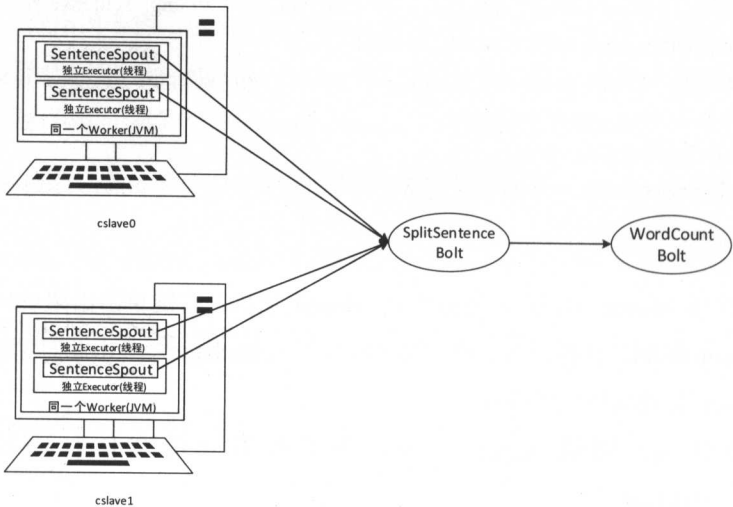


图 10-30 开启多 Worker 模式

当启用 conf.setNumWorkers(2)一句时，该 Topology 不支持本地执行，必须提交到集群。

5. Stream 分组

Stream 分组指的是一个数据流中的 Tuple 如何分发给 Topology 中的不同 Bolt 的 Task，比如在 WCTopology 中，假如系统为 SplitSentenceBolt 类指定了四个 Task，则由 Stream 分组决定特定 Tuple 应分发到哪个 Task 上。当前，Storm 定义了七种数据流分组方式，分别为：

- Shuffle Grouping（随机分组）

随机分发 Tuple 给 Bolt 的各个 Task，每个 Bolt 实例接收的 Tuple 数量大致相同。

- Fields Grouping（按字段分组）

根据指定字段值进行分组，比如说一个数据流根据“word”字段进行分组，则所有具有“word”字段值的 Tuple 会路由到同一个 Bolt 的 Task 中。

- All Grouping（全复制分组）

将所有 Tuple 复制后，发给所有 Bolt 的 Task。每个订阅 Stream 的 Task 都会收到 Tuple 的拷贝。

- Global Grouping（全局分组）

这种分组方式将所有 Tuple 路由到唯一一个 Task 上，该分组方式极易导致某 Task 任务过重而崩溃。

- None Grouping（不分组）

在功能上和随机分组相同，为将来预留。

- Direct Grouping（指向型分组）

数据源会调用 emitDirect() 方法判断一个 Tuple 应该由哪个 Storm 组件来接收，只能在申明为指向型的数据流上使用。

- Local or Shuffle Grouping（本地或随机分组）

和 Shuffle 类似，不过其会尽量将 Tuple 发给同一个 Worker 内的 Bolt Task，该分组可节约网络带宽，提高拓扑性能。

对上述各种分组方式，下面仅给出 Shuffle Grouping 和 All Grouping 示例，限于篇幅，其他分组方式不再讲解。

1) 场景

下面的 WordCountTopology 包含 SentenceSpout、SplitSentenceBolt、WordCountBolt 和 ReportBolt 四个类，且这四个类通过主类 WordCountTopology 的 main 方法构成一本地 Storm-App，下面为这四个类完整代码。

类 SentenceSpout 用于采集原始数据（这里模拟了一个数据源），请注意，编者声明了一个本地线程 ThreadLocal，并在 next() 方法中控制了每个 Thread 只能发出两次变量。


```

public class SentenceSpout extends BaseRichSpout {
    private SpoutOutputCollector collector;
    private String[] sentences = { "aa bb cc", "xx yy zz" };
    private int index = 0;
    private static ThreadLocal<Integer> local = new ThreadLocal<Integer>() {
        public Integer initialValue() {
            return 0;
        }
    };
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("sentence"));
    }
    public void open(Map config, TopologyContext context, SpoutOutputCollector collector) {
        this.collector = collector;
    }
    public void nextTuple() {
        if (local.get() >= 2) {return;}
        this.collector.emit(new Values(sentences[index]));
        System.out.println("AAA "+Thread.currentThread().getName()+" "+this+" "+sentences[index]);
        index++;
        if (index >= sentences.length) {index = 0;}
        local.set(local.get() + 1);
        try {TimeUnit.MICROSECONDS.sleep(100);} catch (InterruptedException e) {e.printStackTrace();}
    }
}

```

SplitSentenceBolt 和原来的相同，无须做任何更改，不过，编者在 execute()里添加了打印输出语句：

```

public class SplitSentenceBolt extends BaseRichBolt{
    private OutputCollector collector;
    public void prepare(Map config, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
    }
    public void execute(Tuple tuple) {
        String sentence = tuple.getStringByField("sentence");
        System.out.println("BBB "+Thread.currentThread().getName()+" "+this+" "+sentence);
        String[] words = sentence.split(" ");
        for(String word : words){this.collector.emit(new Values(word));}
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}

```

类 WordCountBolt 用于实时统计单词个数，其代码如下：

```

public class WordCountBolt extends BaseRichBolt {
    private OutputCollector collector;
    private HashMap<String, Long> counts = null;
    public void prepare(Map config, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
        this.counts = new HashMap<String, Long>();
    }
    public void execute(Tuple tuple) {
        String word = tuple.getStringByField("word");
        Long count = this.counts.get(word);
        if(count == null){ count = 0L; }
        count++;
        this.counts.put(word, count);
        this.collector.emit(new Values(word, count));
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}

```

类 `ReportBolt` 用于观测最终结果，在 `cleanup()` 方法里，编者写明了程序结束时打印输出语句，请注意该方法只有本地测试时有效，在集群上执行时，该方法失效：

```

public class ReportBolt extends BaseRichBolt {
    private HashMap<String, Long> counts = null;
    public void prepare(Map config, TopologyContext context, OutputCollector collector) {
        this.counts = new HashMap<String, Long>();
    }
    public void execute(Tuple tuple) {
        String word = tuple.getStringByField("word");
        Long count = tuple.getLongByField("count");
        this.counts.put(word, count);
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        // this bolt does not emit anything
    }
    @Override
    public void cleanup() {
        System.out.println("--- Final Counts ---");
        List<String> keys = new ArrayList<String>();
        keys.addAll(this.counts.keySet());
        Collections.sort(keys);
        for (String key : keys) {
            System.out.println(key + " : " + this.counts.get(key));
        }
    }
}

```



```

        System.out.println("----- App End -----");
    }
}

```

类 WordCountTopology 用来定义整个 Topology,

```

public class WordCountTopology {
    public static void main(String[] args) throws Exception {
        SentenceSpout spout = new SentenceSpout();
        SplitSentenceBolt splitBolt = new SplitSentenceBolt();
        WordCountBolt countBolt = new WordCountBolt();
        ReportBolt reportBolt = new ReportBolt();
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("sentence-spout", spout, 2).setNumTasks(2);
        // SentenceSpout --> SplitSentenceBolt
        builder.setBolt("split-bolt",
splitBolt, 4).setNumTasks(4).shuffleGrouping("sentence-spout");
        //builder.setBolt("split-bolt", splitBolt, 4).setNumTasks(4).allGrouping("sentence-spout");
        // SplitSentenceBolt --> WordCountBolt
        builder.setBolt("count-bolt", countBolt).fieldsGrouping("split-bolt", new Fields("word"));
        // WordCountBolt --> ReportBolt
        builder.setBolt("report-bolt", reportBolt).globalGrouping("count-bolt");
        Config config = new Config();
        LocalCluster cluster = new LocalCluster();
        cluster.submitTopology("word-count-topology", config, builder.createTopology());
        waitForSeconds(8);
        cluster.killTopology("word-count-topology");
        cluster.shutdown();
    }
}

```

上述 main 方法的 builder.setSpout("sentence-spout", spout, 2).setNumTasks(2) 一句指明本 WCTopology 申请了两个 Executor 来执行两个 SentenceSpout，即 WCTopology 拥有两个 Executor，且每个 Executor 执行一个 SentenceSpout（图 10-31）。

事实上第一个“2”指的是 Executor 个数，而在 Storm 中可以将 Executor 概念理解为线程，Executor 数量理解成线程数，后一个“2”指的是 SentenceSpout 的数量。此行代码正确理解是：WCTopology 须启动两个 SentenceSpout（后一个 2），WCTopology 向 Nimbus 申请两个 Executor（前一个 2），故每个 Executor（一个 Executor 相当于一个独立线程）上执行一个 SentenceSpout。

2) ShuffleGrouping

当使用如下代码时，即启用 shuffleGrouping:

```

builder.setBolt("split-bolt", splitBolt, 4).setNumTasks(4).shuffleGrouping("sentence-spout");

```

```
//builder.setBolt("split-bolt", splitBolt,4).setNumTasks(4).allGrouping("sentence-spout");
```

该句指的是 WCTopology 申请了四个 Executor 来执行四个 SentenceSpout，即 WCTopology 拥有四个 Executor，且每个 Executor 执行一个 SentenceSpout（图 10-31）。

程序开发结束，为确保项目已编译，不妨左键选中该 Maven 项目，右键单击出菜单项，依次进入 Run As→ Maven test。最后，左键选中类 WordCountTopology，右键单击出菜单项，依次进入 Run As→ Java Application。程序执行 8 秒后，将自动退出并输出如下结果：

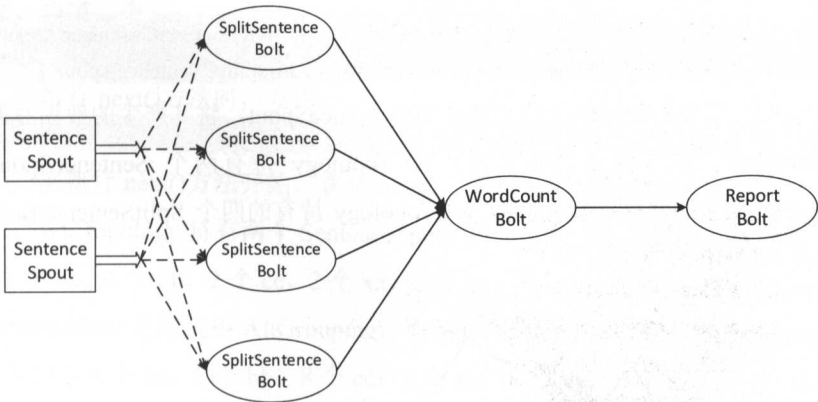


图 10-31 Shuffle Grouping 示例

```
--- Final Counts ---
aa : 2
bb : 2
cc : 2
xx : 2
yy : 2
zz : 2
----- App End -----
```

显然，此时 WCTopology 使用 Shuffle Grouping，且其共有持有两个 SentenceSpout 对象，而每个 SentenceSpout 对象都受到 if (local.get() >= 2) {return;}控制，故当第一个 SentenceSpout 对象第一次执行 next()方法时，只发出：

```
"aa bb cc"
```

第二次执行 next()方法时，发出：

```
"xx yy zz"
```

从第三次执行 next()方法开始，直接 return，不再发出任何数据。

同理，当第二个 SentenceSpout 对象第一次执行 next()方法时，发出：

```
"aa bb cc"
```

第二次执行 next()方法时，发出：

```
"xx yy zz"
```

从第三次执行 next() 方法开始，直接 return，不再发出任何数据。

这样，WCTopology 持有两个 SentenceSpout 共计发送了四条数据，拆分成一个个单词则为：2 个 aa，2 个 bb，2 个 cc，2 个 xx，2 个 yy，2 个 zz。由于这两个 SentenceSpout 和 SplitSentenceBolt 之间采用 shuffleGrouping，故最终结果也是 2 个 aa，2 个 bb，2 个 cc，2 个 xx，2 个 yy，2 个 zz。

3) AllGrouping

当使用如下代码时，即启用 allGrouping：

```
//builder.setBolt("split-bolt", splitBolt,4).setNumTasks(4).shuffleGrouping("sentence-spout");
builder.setBolt("split-bolt", splitBolt,4).setNumTasks(4).allGrouping("sentence-spout");
```

该句表明，对 WCTopology 启动的所有 SentenceSpout，将其产生的数据复制至所有 SplitSentenceBolt。在本例中的意义为：WCTopology 持有两个 SentenceSpout，每个 SentenceSpout 发送的数据都要复制至 WCTopology 持有的四个 SplitSentenceBolt 里，其执行示意图如图 10-32 所示。

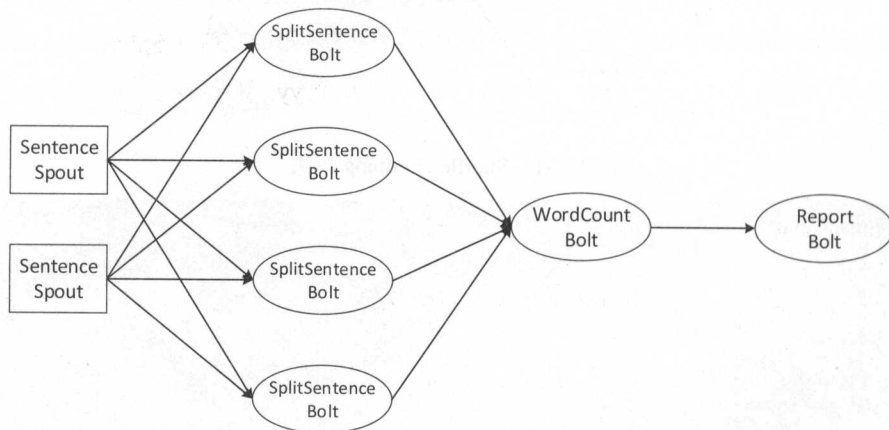


图 10-32 AllGrouping 分组示例

程序开发结束，为确保项目已编译，请左键选中该 Maven 项目，右键单击出菜单项，依次进入 Run As→Maven test。最后，左键选中类 WordCountTopology，右键单击出菜单，依次进入 Run As→Java Application。程序执行 8 秒后，将自动退出并输出如下结果：

```
--- Final Counts ---
aa : 8
bb : 8
cc : 8
xx : 8
yy : 8
zz : 8
----- App End -----
```

显然,此时 WCTopology 使用 All Grouping, 且其共有持有两个 SentenceSpout 对象, 而每个 SentenceSpout 对象都受到 `if (local.get() >= 2) {return;}` 控制, 故当第一个 SentenceSpout 对象第一次执行 `next()` 方法时, 只发出:

```
"aa bb cc"
```

第二次执行 `next()` 方法时, 发出:

```
"xx yy zz"
```

从第三次执行 `next()` 方法开始, 直接 `return`, 不再发出任何数据。

同理, 当第二个 SentenceSpout 对象第一次执行 `next()` 方法时, 发出:

```
"aa bb cc"
```

第二次执行 `next()` 方法时, 发出:

```
"xx yy zz"
```

从第三次执行 `next()` 方法开始, 直接 `return`, 不再发出任何数据。

综上, WCTopology 持有两个 SentenceSpout 共计发送了四条数据, 拆分成一个个单词则为: 2 个 aa, 2 个 bb, 2 个 cc, 2 个 xx, 2 个 yy, 2 个 zz。由于这两个 SentenceSpout 和 SplitSentenceBolt 之间采用 AllGrouping, 而 WCTopology 持有四个 SplitSentenceBolt, 故最终结果也是 8 个 aa, 8 个 bb, 8 个 cc, 8 个 xx, 8 个 yy, 8 个 zz。

10.2 Storm 接口

Storm 接口指的是用户取得 Storm 服务的途径, 下面先讲述常见的 Storm 接口, 接着直接讲述 Web 接口。

1. 接口汇总

作为大数据处理最常用的实时计算框架, 针对不同的上层应用, Storm 框架提供了四类统一访问接口, 分别为:

- Storm 自带 Web 接口
- Storm Shell 接口
- Storm API 接口
- Trident API 接口

Web 接口主要为管理员提供, 从该页面上, 管理员能查看当前系统中所有 Storm-App, 通过该页面, 管理员还可以终止正在运行的 Storm-App, 不过该页面只支持读、不支持写操作。

Shell 接口主要针对 Storm 管理员和程序员，通过 Shell 接口，程序员能够向 Storm 集群提交 Storm-App、查看正在运行的 Storm-App，10.3 节将重点 Storm 的 Shell 接口。

Storm API 面向 Java 和 Clojure 工程师，程序员可以通过该接口编写 Storm-App 用户层代码 Spout 和 Bolt，10.4 节将讲述 Storm 编程。

Trident 在 Storm 上提供了高层抽象，提供了大量实用功能以支撑函数操作，限于篇幅，本章不讲述 Trident。

2. 实战 Storm Web

由于 Storm Web 接口内容较少，此处直接讲解。Storm UI 的默认地址是“StormUIIP:8080”，由于此处 UI 所在机为 cmaster0，故在启动 Storm 集群后（UI 进程也须启动），浏览器打开“cmaster0:808”，即可进入 Storm 默认 Web 界面（图 10-33）。从该界面上，可看到集群资源统计信息、应用程序列表、Supervisor 节点列表等。



图 10-33 Storm UI 界面

和图 10-33 对应，图 10-34 为正在运行两个 Topology 的 Storm UI 界面，点击“wc-0”，即可进入名为“wc-0”的 Storm-App 的统计页面（图 10-35）。

从“wc-0”统计页面上，程序员可终止本应用，查看本应用使用资源量，查看本 Topology 包含 Spout 和 Bolt 数量等操作，图 10-36 为点击 Spout 后，进入的 Spout 统计页面。

Storm UI

Cluster Summary

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.10.0	1h 57m 58s	4	6	10	16	46	46

Topology Summary

Name	Id	Owner	Status	Uptime	Num workers	Num executors	Num tasks	Scheduler Info
ec-0	ec-0-7-1461002873		ACTIVE	49s	3	18	18	
wc-0	wc-0-8-1461002908		ACTIVE	14s	3	28	28	

图 10-34 正在运行两个 Topology 的 Storm UI 界面

Storm UI

Topology summary

Name	Id	Owner	Status	Uptime	Num workers	Num executors	Num tasks	Scheduler Info
wc-0	wc-0-8-1461002908		ACTIVE	7m 38s	3	28	28	

Topology actions

Activate Deactivate Rebalance Kill

Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	297460	159460	0.000	0	0
3h 0m 0s	297460	159460	0.000	0	0
1d 0h 0m 0s	297460	159460	0.000	0	0
All time	297460	159460	0.000	0	0

Spouts (All time)

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Error Host	Error Port	Last error
spout	5	5	21540	21540	0.000	0	0			

Bolts (All time)

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Error Host	Error Port	Last error
count	12	12	138000	0	0.013	0.217	137880	0.112	137960	0			
split	8	8	137920	137920	0.001	0.036	21580	8.688	21520	0			

图 10-35 “wc-0” Topology 详细信息

StormWeb 接口功能简单实用，主要用于统计集群资源，罗列并统计当前作业数，为管理员管理 Storm-App 带来诸多便利。

Storm UI

Component summary

Id	Topology	Executors	Tasks
epout	wc-0	5	5

Spout stats

WordCountSpout统计数据

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	29540	29540	0.000	0	0
3h 0m 0s	73380	73380	0.000	0	0
1d 0h 0m 0s	73380	73380	0.000	0	0

Output stats (All time)

Stream	Emitted	Transferred	Complete latency (ms)	Acked	Failed
default	73390	73380	0	0	0

Showing 1 to 1 of 1 entries

Executors (All time)

执行WordCountSpout的各Executor统计

Id	Uptime	Host	Port	Emitted	Transferred	Complete latency (ms)	Acked	Failed
[24-24]	24m 50s	cslave1	6701	14690	14680	0.000	0	0
[25-25]	24m 49s	cslave3	6700	14840	14640	0.000	0	0
[26-26]	24m 45s	cslave2	6701	14720	14720	0.000	0	0
[27-27]	24m 50s	cslave1	6701	14680	14680	0.000	0	0
[28-28]	24m 49s	cslave3	6700	14690	14660	0.000	0	0

图 10-36 “wc-0” 的 Spout 统计信息

10.3 实战 Storm Shell

Storm Shell 接口是 Storm 功能的实际体现，用户可以使用 Storm Shell 命令完成集群管理和任务管理（图 10-37）。

```
[allen@iclient0 apache-storm-0.10.0]$ bin/storm
Commands:
  activate
  classpath
  deactivate
  dev-zookeeper
  drpc
  help
  jar
  kill
  list
  localconfvalue
  logviewer
  monitor
  nimbus
  rebalance
  remoteconfvalue
  repl
  shell
  supervisor
  ui
  upload-credentials
  version

Help:
  help
  help <command>

Documentation for the storm client can be found at http://storm.incubator.apache.org/documentation/Command-line-client.html

Configs can be overridden using one or more -c flags, e.g. "storm list -c ni
mbus.host=nimbus.mycompany.com"

[allen@iclient0 apache-storm-0.10.0]$
```

Storm命令行统一入口

图 10-37 Storm Shell 统一入口

1. list

该命令用来罗列 Storm 集群中所有 Topology，图 10-38 为命令执行效果图，从图中可以看出，list 命令会给出当前 Topology 的运行状态。

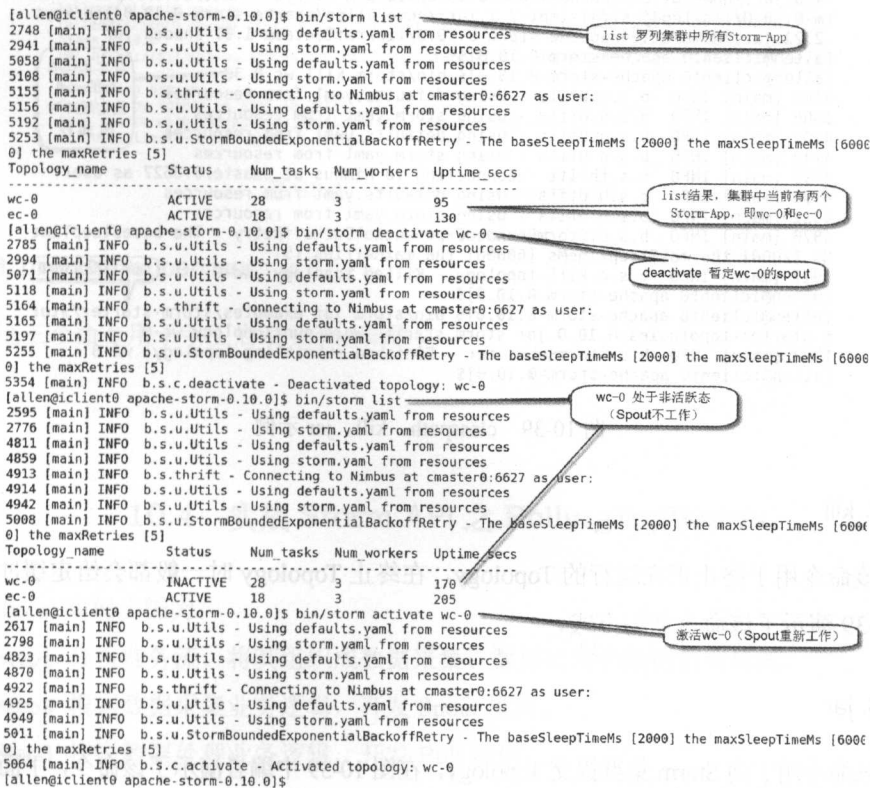


图 10-38 list、activate、deactivate 示例

2. activate、deactivate

deactivate 用于将 Topology 的 Spout 设置成“非活跃状态”，activate 命令则正好相反。由于 Spout 是 Topology 中的数据采集器，故其处于“非活跃状态”时，整个 Topology 也就停止了，图 10-38 中演示了 activate 和 deactivate 用法。

3. classpath

该命令用于显示 Storm 完整环境变量，程序员可能需要该变量进行代码环境设置，图 10-39 为该命令执行示意图。

```

[allen@iclient0 apache-storm-0.10.0]$ bin/storm classpath
/home/allen/apache-storm-0.10.0/lib/log4j-over-slf4j-1.6.6.jar:/home/allen/apac
he-storm-0.10.0/lib/storm-core-0.10.0.jar:/home/allen/apache-storm-0.10.0/lib/c
lojure-1.6.0.jar:/home/allen/apache-storm-0.10.0/lib/log4j-api-2.1.jar:/home/al
len/apache-storm-0.10.0/lib/slf4j-api-1.7.7.jar:/home/allen/apache-storm-0.10.0
/lib/minlog-1.2.jar:/home/allen/apache-storm-0.10.0/lib/disruptor-2.10.4.jar:/h
ome/allen/apache-storm-0.10.0/lib/servlet-api-2.5.jar:/home/allen/apache-storm-
0.10.0/lib/log4j-core-2.1.jar:/home/allen/apache-storm-0.10.0/lib/hadoop-auth-2
.4.0.jar:/home/allen/apache-storm-0.10.0/lib/asm-4.0.jar:/home/allen/apache-sto
rm-0.10.0/lib/log4j-slf4j-impl-2.1.jar:/home/allen/apache-storm-0.10.0/lib/kryo
-2.21.jar:/home/allen/apache-storm-0.10.0/lib/reflectasm-1.07-shaded.jar
[allen@iclient0 apache-storm-0.10.0]$
[allen@iclient0 apache-storm-0.10.0]$ bin/storm kill wc-0 30
3160 [main] INFO b.s.u.Utils - Using defaults.yaml from resources
3306 [main] INFO b.s.u.Utils - Using storm.yaml from resources
4705 [main] INFO b.s.u.Utils - Using defaults.yaml from resources
4749 [main] INFO b.s.u.Utils - Using storm.yaml from resources
4790 [main] INFO b.s.thrift - Connecting to Nimbus at cmaster0:6627 as user:
4790 [main] INFO b.s.u.Utils - Using defaults.yaml from resources
4821 [main] INFO b.s.u.Utils - Using storm.yaml from resources
4876 [main] INFO b.s.u.StormBoundedExponentialBackoffRetry - The baseSleepTime
Ms [2000] the maxSleepTimeMs [60000] the maxRetries [5]
4917 [main] INFO b.s.c.kill-topology - Killed topology: wc-0
[allen@iclient0 apache-storm-0.10.0]$
[allen@iclient0 apache-storm-0.10.0]$ bin/storm jar examples/storm-starter/stor
m-starter-topologies-0.10.0.jar storm.starter.WordCountTopology wc-0
1792 [main] INFO b.s.StormSubmitter - Finished submitting topology: wc-0
[allen@iclient0 apache-storm-0.10.0]$

```

终止正在运
行的wc-0
给定缓冲时
间 30 秒

向集群提交wc-0

图 10-39 classpath、kill、jar 示例

4. kill

该命令用于终止正在运行的 Topology，在终止 Topology 时一般都会给定缓冲时间，图 10-39 演示了该命令使用方式。

5. jar

该命令用于向 Storm 集群提交 Topology，在图 10-39 中编者演示了该命令，下面这两条命令也是使用 jar 命令向 Storm 集群提交 Topology，其中第一条为提交 WordCountTopology，第二条为提交 ExclamationTopology。

```

[allen@iclient0 apache-storm-0.10.0]$ bin/storm jar examples/storm-starter/storm-starter-
topologies-0.10.0.jar storm.starter.WordCountTopology wc-0
[allen@iclient0 apache-storm-0.10.0]$ bin/storm jar examples/storm-starter/storm-starter-
topologies-0.10.0.jar storm.starter.ExclamationTopology ec-0

```

6. rebalance

该命令用于将指定 Topology 的 Worker 均匀分散于集群各节点，假定你的 Storm 集群中有 10 个节点，当前正在运行一个 Topology (Worker 数量大于 10)，现在你新添加了 5 个节点，则 rebalance 命令可以在不关闭 Topology 的情况下，使之均衡化 (图 10-40)。

7. version

显示 Storm 版本 (图 10-40)。

```

[allen@iclient0 apache-storm-0.10.0]$ bin/storm rebalance wc-0 -w 30
3356 [main] INFO b.s.u.Utils - Using defaults.yaml from resources
3505 [main] INFO b.s.u.Utils - Using storm.yaml from resources
5004 [main] INFO b.s.u.Utils - Using defaults.yaml from resources
5047 [main] INFO b.s.u.Utils - Using storm.yaml from resources
5108 [main] INFO b.s.thrift - Connecting to Nimbus at cmaster0:6627 as user:
5109 [main] INFO b.s.u.Utils - Using defaults.yaml from resources
5140 [main] INFO b.s.u.Utils - Using storm.yaml from resources
5195 [main] INFO b.s.u.StormBoundedExponentialBackoffRetry - The baseSleepTimeMs [2000] the
maxSleepTimeMs [60000] the maxRetries [5]
5251 [main] INFO b.s.c.rebalance - Topology wc-0 is rebalancing
[allen@iclient0 apache-storm-0.10.0]$ bin/storm version
Storm 0.10.0
URL https://git-wip-us.apache.org/repos/asf/storm.git -r d02f94268dec229d1125a24fdf53fa303cbc
2b29
Branch (no branch)
Compiled by tgoetz on 2015-10-23T19:23Z
From source with checksum 371dfb55b490dd397f27d81b90b69fe
[allen@iclient0 apache-storm-0.10.0]$

```

rebalancer命令
均衡化wc-0各Worker

结果

查看Storm版本

图 10-40 rebalance、version 示例

8. nimbus、supervisor、ui

这三个命令用于手工启动 Storm 集群。

10.4 实战 Storm API 之 RollingTopWords

Storm 编程步骤为：

Step1 分析业务流，将其组织成数据采集、数据消费和数据消费模式。

Step2 根据数据采集业务逻辑，开发 Spout 模块。

Step3 根据数据处理业务逻辑，开发 Bolt 模块。

Step4 根据数据消费业务逻辑，开发 Bolt 模块。

Step5 编写主函数，将各模块连接起来。

Storm 的 example 在 “apache-storm-0.10.0/examples/storm-starter/src/jym/storm/starter” 目录下，本节讲述的 RollingTopWords 也在该目录。这些代码都是 Storm 的入门代码，建议读者学习该目录下所有代码后，在学习 Trident。

RollingTopWords 为 Storm 自带的示例应用，同 WordCount 一样，是学习 Storm 编程最基础的代码。RollingTopWords 主要用于实时统计前 N 个出现次数最多的单词，其由 TestWordSpout、RollingCountBolt、IntermediateRankingsBolt、TotalRankingsBolt 四个模块构成。其中 TestWordSpout 用于生成数据，RollingCountBolt 用于统计单词出现次数，IntermediateRankingsBolt 用于实时排序，TotalRankingsBolt 用于显示排序最高的那几个单词。

由于 RollingTopWords 代码量太大，下面只给出了其主方法，关于 TestWordSpout、RollingCountBolt、IntermediateRankingsBolt、TotalRankingsBolt 请参考 Storm 自带的源码。

```

package storm.starter;
import backtype.storm.Config;
import backtype.storm.testing.TestWordSpout;
import backtype.storm.topology.TopologyBuilder;
import backtype.storm.tuple.Fields;
import org.apache.log4j.Logger;
import storm.starter.bolt.IntermediateRankingsBolt;
import storm.starter.bolt.RollingCountBolt;
import storm.starter.bolt.TotalRankingsBolt;
import storm.starter.util.StormRunner;
public class RollingTopWords {
    private static final Logger LOG = Logger.getLogger(RollingTopWords.class);
    private static final int DEFAULT_RUNTIME_IN_SECONDS = 60;
    private static final int TOP_N = 5;
    private final TopologyBuilder builder;
    private final String topologyName;
    private final Config topologyConfig;
    private final int runtimeInSeconds;
    public RollingTopWords(String topologyName) throws InterruptedException {
        builder = new TopologyBuilder();
        this.topologyName = topologyName;
        topologyConfig = createTopologyConfiguration();
        runtimeInSeconds = DEFAULT_RUNTIME_IN_SECONDS;
        wireTopology();
    }
    private static Config createTopologyConfiguration() {
        Config conf = new Config();
        conf.setDebug(true);
        return conf;
    }
    private void wireTopology() throws InterruptedException {
        String spoutId = "wordGenerator";
        String counterId = "counter";
        String intermediateRankerId = "intermediateRanker";
        String totalRankerId = "finalRanker";
        builder.setSpout(spoutId, new TestWordSpout(), 5);
        builder.setBolt(counterId, new RollingCountBolt(9, 3), 4).fieldsGrouping(spoutId, new
Fields("word"));
        builder.setBolt(intermediateRankerId, new
        IntermediateRankingsBolt(TOP_N, 4).fieldsGrouping(counterId, new Fields("obj")));
        builder.setBolt(totalRankerId, new
        TotalRankingsBolt(TOP_N)).globalGrouping(intermediateRankerId);
    }
    public void runLocally() throws InterruptedException {

```



```

StormRunner.runTopologyLocally(builder.createTopology(), topologyName, topologyConfig, runtimeInSeconds);
    }
    public void runRemotely() throws Exception {
        StormRunner.runTopologyRemotely(builder.createTopology(), topologyName, topologyConfig);
    }
    public static void main(String[] args) throws Exception {
        String topologyName = "slidingWindowCounts";
        if (args.length >= 1) {
            topologyName = args[0];
        }
        boolean runLocally = true;
        if (args.length >= 2 && args[1].equalsIgnoreCase("remote")) {
            runLocally = false;
        }
        LOG.info("Topology name: " + topologyName);
        RollingTopWords rtw = new RollingTopWords(topologyName);
        if (runLocally) {
            LOG.info("Running in local mode");
            rtw.runLocally();
        }
        else {
            LOG.info("Running in remote (cluster) mode");
            rtw.runRemotely();
        }
    }
}

```

习 题

1. 简述 Storm 和 Hadoop、Spark 的区别联系。
2. 简述 Storm 框架功能作用及其体系架构。
3. Storm 采用什么机制确保 Tuple 不丢失？
4. 简述手工部署 Storm、使用 Ambari 部署 Storm 的步骤。
5. 简述 Storm 访问接口。
6. 简述使用 Maven 和不使用 Maven 时，Storm 开发环境搭建步骤。
7. 简述 Storm-App (Topology) 编程步骤和执行步骤。
8. 在大型系统中，如何使用 Storm 来提供实时服务？
9. 显然，Storm 适合处理增量数据，在大型系统中，使用 Storm 处理增量数据的方

式和 MapReduce 有何区别?

10. 在大型集群中, 如何根据单机配置, 对 Topology 设置不同的并行度?
11. 在大型集群中, 如何实时控制 Topology 运行占用的系统资源量?
12. 在一个大型集群中, 对于一个长期运行的 Topology, 如何重新实现资源再平衡?

参考文献

- [1] <http://storm.apache.org/>
- [2] <https://storm.apache.org/javadoc/apidocs/index.html>
- [3] <http://storm.apache.org/tutorial.html>
- [4] <http://storm.apache.org/documentation/Setting-up-a-Storm-cluster.html>

图 11-1-1 展示了 Hive 的架构。Hive 是一个建立在 Hadoop 之上的数据仓库工具，它可以将结构化的数据文件存储在 Hadoop 分布式文件系统（HDFS）中，并使用 MapReduce 或 Tez 引擎进行查询和计算。Hive 的架构可以分为三层：客户端层、Hive 服务层和 Hadoop 层。客户端层包括 Hive CLI、Hive JDBC、Hive ODBC 等；Hive 服务层包括 Hive Metastore、Hive Server、Hive Driver 等；Hadoop 层包括 HDFS、MapReduce、Tez 等。

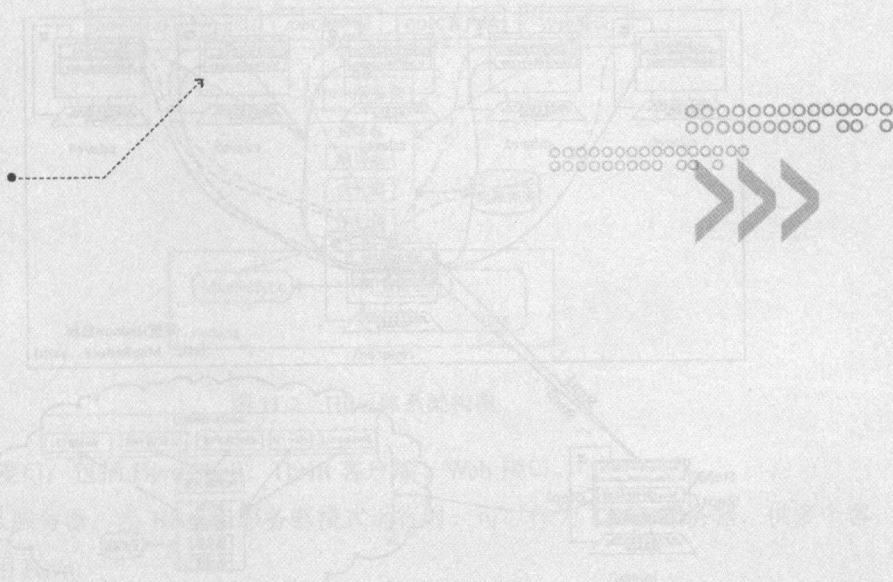
图 11-1-2 展示了 Hive 的部署架构。Hive 可以部署在 Hadoop 集群上，也可以部署在云平台上。Hive 的部署架构可以分为两种：集中式和分布式。集中式部署是指将 Hive 的所有组件都部署在同一个节点上；分布式部署是指将 Hive 的组件分布在不同的节点上。Hive 的部署架构还可以分为本地部署和远程部署。本地部署是指将 Hive 部署在本地机器上；远程部署是指将 Hive 部署在远程服务器上。

第 11 章

数据仓库工具 Hive

HADOOP

BEING DIGITAL



Hive 是 Hadoop 大数据生态圈中的数据仓库，其提供以表格的方式来组织与管理 HDFS 上的数据、以类 SQL 的方式来操作表格里的数据。本章在介绍 Hive^[1]的工作原理及其体系架构后，将重点讲述编写 HiveQL 完成大数据分析。

11.1 Hive 简介

Hive 是一个构建在 Hadoop 上的数据仓库框架，它起源于 Facebook 内部信息处理平台。由于需要处理大量社会网络数据，考虑到扩展性，Facebook 最终选择 Hadoop 作为存储和处理平台。Hive 的设计目的是让 Facebook 内精通 SQL（但 Java 编程相对较弱）的分析师能够以类 SQL 的方式查询存放在 HDFS 的大规模数据集。

11.1.1 工作原理

Hive 非常简单，本质上，其相当于一个 MapReduce 和 HDFS 的翻译终端。用户提交 Hive 脚本后，Hive 运行时环境会将这些脚本翻译成 MapReduce 和 HDFS 操作并向集群提交这些操作。

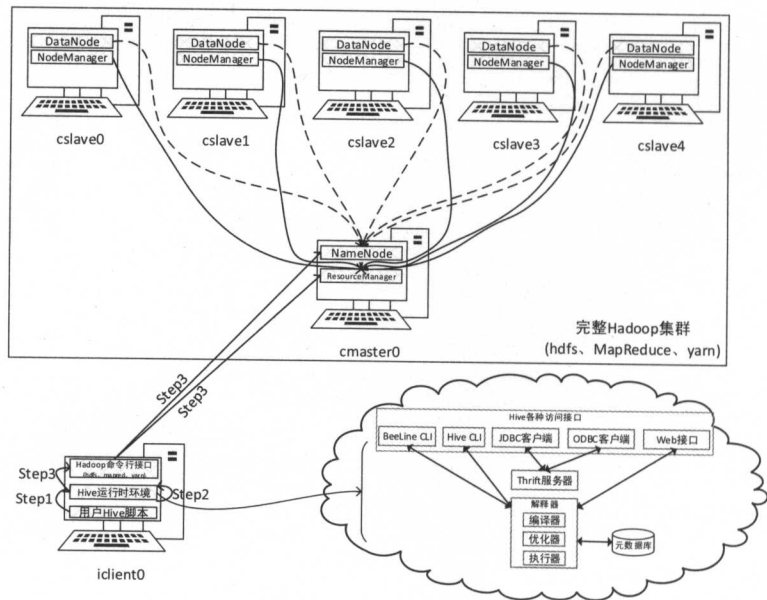


图 11-1 Hive 工作原理图

当用户向 Hive 提交其编写的 HiveQL 后，首先，Hive 运行时环境会将这些脚本翻译成 MapReduce 和 HDFS 操作；紧接着，Hive 运行时环境使用 Hadoop 命令行接口向 Hadoop 集群提交这些 MapReduce 和 HDFS 操作；最后，Hadoop 集群逐步执行这些 MapReduce 和 HDFS 操作，整个过程可概括如下：

Step1 用户编写 HiveQL 并向 Hive 运行时环境提交该 HiveQL（图 11-1 中 Step1）。

Step2 Hive 运行时环境将该 HiveQL 翻译成 MapReduce 和 HDFS 操作（图 11-1 中 Step2）。

Step3 Hive 运行时环境调用 Hadoop 命令行接口或程序接口，向 Hadoop 集群提交翻译后的 HiveQL（图 11-1 中 Step3）。

Step4 Hadoop 集群执行 HiveQL 翻译后的 MapReduce-App 或 HDFS-App（图 11-1 中未标）。

由上述执行过程可知，Hive 的核心是其运行时环境，该环境能够将类 SQL 语句翻译成 MapReduce，下面的体系架构一节，即深入 Hive 内部，剖析其执行机制。

11.1.2 体系架构

Hive 主要包含 Shell 环境、元数据库、解析器等组件^[2]（图 11-2），按功能，可将这些组件分成如下几大模块。

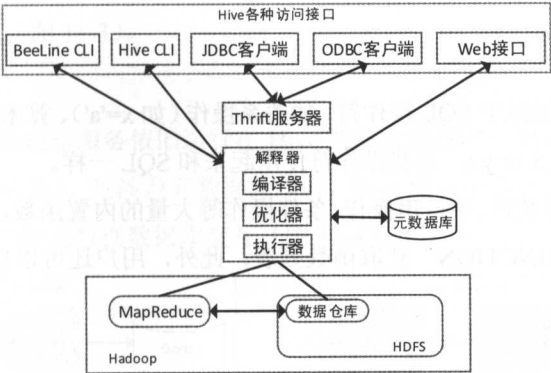


图 11-2 Hive 体系架构图

- ①用户接口：包括 Hive Shell、Thrift 客户端、Web 接口。
- ②Thrift 服务器：当 Hive 以服务器模式运行时，可以作为 Thrift 服务器，供多个客户端同时使用 Hive。
- ③元数据库：Hive 元数据（如表信息）的集中存放地。

④解析器: 包括解释器、编译器、优化器、执行器。其是将 HiveQL 翻译成 MapReduce 和 HDFS 的核心部件。

⑤Hadoop: 底层分布式存储和计算引擎。

需要注意的是, 上述的 Hadoop 集群并不属于 Hive 运行时环境。实际上, Hive 就相当于 Hadoop 的一个智能终端, 该终端的核心作用是支持编写、翻译, 并 (向 Hadoop 集群) 提交类 SQL 的大数据分析语句。

11.1.3 计算模型

Hive 的 SQL 称为 HiveQL, 它与大部分的 SQL 语法兼容。不过, 由于无论中间怎么“折腾”, 其最终都是翻译成 MapReduce 和 HDFS, 其必然有独特之处, 比如 HiveQL 不支持更新操作, 再比如 HiveQL 中有 MAP 和 REDUCE 子句等。目前, Hive 上已开发的函数集可满足绝大部分需求, 下面简介 Hive 数据类型与常用函数, 至于 Hive 表类型、桶和分区等这里不深入介绍。

1. 数据类型

Hive 支持基本类型和复杂类型, 基本类型主要有数值型、布尔型和字符串, 复杂类型为 ARRAY、MAP 和 STRUCT。

2. 操作和函数

HiveQL 操作符类似于 SQL 操作符, 如关系操作 (如 $x='a'$)、算术操作 (如加法 $x+1$)、逻辑操作 (如逻辑或 $x \text{ or } y$), 这些操作符使用起来和 SQL 一样。

Hive 提供了数理统计、字符串操作、条件操作等大量的内置函数, 用户可在 Hive Shell 端中输入 “SHOW FUNCTION” 获取函数列表, 此外, 用户还可以自己编写函数。

3. 计算模型实例

以下为一个 Hive 计算模式实例, 显然, 其建表语音和 SQL 非常类似, 简单明了。唯一不同的是, 该建表语句的最后一行指定了分隔符和存储格式, 这是由于在 HDFS 中存储数据时, 须指定字段间分割符和存储格式。

```
CREATE TABLE u_data(
  userid INT,
  movieid INT,
  rating INT,
```

```
unixtime STRING
)ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE;

待建好表后，可使用下述语句 Select 语句，查询该表中的指定数据。

SELECT COUNT(*) FROM u_data;
```

由于新建的 u_data 中并无数据，上述 select 语句执行结果应当为空，编者将会在实战环节再次讲述该语句。

11.1.4 集群部署

相对于其他组件，Hive 部署要复杂得多，按 metastore 存储位置的不同，其部署模式分为内嵌模式、本地模式和完全远程模式三种。当使用完全模式时，可以提供很多用户同时访问并操作 Hive，并且此模式还提供各类接口（BeeLine，CLI，甚至是 Pig），下面简单介绍这三种模式。

1. Hive 部署模式

1) 内嵌模式

此模式是安装时的默认部署模式，此时元数据存储在一个内嵌数据库 Derby 中，并且所有组件（如数据库、元数据服务）都运行在同一个进程内，这种模式下，一段时间内只支持一个活动用户。但这种模式配置简单，所需机器较少，限于集群规模，本节 Hive 部署即采用这种模式（图 11-3）。

2) 本地模式

此模式是 Hive 元数据服务依旧运行在 Hive 服务主进程中，但元数据存储在一个独立的数据库里（可以是远程机器），当涉及元数据操作时，Hive 服务中的元数据服务模块会通过 JDBC 和存储于 DB 里的元数据数据库交互（图 11-4）。



图 11-3 内嵌模式示例



图 11-4 本地模式示例

3) 完全远程模式

此时，元数据服务以独立进程运行，并且元数据存储在一个独立的数据库里。此时 HiveServer2，Hhatalog，Pig 等其他进程可以使用 Thrift 客户端通过网络来获取元数据服务。而 metastore service 则通过 JDBC 和存储在数据库（如 MySQL）里的 metastore database

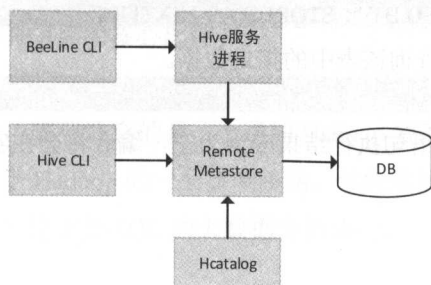


图 11-5 完全远程模式示例

交互。其实，这也是典型的网站架构模式，前台页面给出查询语句，中间层使用 Thrift 网络 API 将查询传到 Metastore service，接着 Metastore service 根据查询得出相应结果，并给出回应（图 11-5），littleCstor 中的 Hive 即属于该模式。

2. 内嵌模式部署

由于使用内嵌模式时，其 Hive 会使用内置的 Derby 数据库来存储数据库，显然此时无须考虑数据库部署连接问题，因此该部署过程非常简单，只要在 iclient0 解压 Hive 包并配置 Hadoop 环境变量即可，整个部署过程可概括如下。

Step1 下载并解压 Hive。

读者可使用熟悉的下载工具，下载 Hive 安装包，不过不管用何工具，最终都要将 Hive 拷至 iclient0 上。待拷贝到 iclient0 上后，可使用下述命令解压 Hive：

```
[allen@iclient0 ~]$ tar -zxvf /home/allen/apache-hive-1.2.1-bin.tar.gz #iclient0 机,allen 用户
```

当前 Hive 的稳定版为 1.2.1，故编者也使用该稳定版，上述完成以 allen 身份，在 iclient0 上，将“/home/allen”目录下的 apache-hive-1.2.1-bin.tar.gz 解压到“/home/allen”目录下。

Step2 为 Hive 配置 Hadoop 安装路径。

待解压完后，进入 Hive 的配置文件夹 conf，接着将 Hive 的环境变量模板文件复制成环境变量文件。

```
[allen@iclient0 ~]$ cd apache-hive-1.2.1-bin/conf/
[allen@iclient0 conf]$ cp hive-env.sh.template hive-env.sh
[allen@iclient0 conf]$ vim hive-env.sh
```

上述的 hive-env.sh 即为 Hive 默认环境文件，读者可使用 vim 等命令，编辑并将 Hadoop 环境变量写入该文件。比如编者的 Hadoop 部署位置为“/home/allen/hadoop-2.7.1”，故实际操作中，编者将下述语句加入了 hive-env.sh：

```
HADOOP_HOME=/home/allen/hadoop-2.7.1
```

图 11-6 为编者编辑 hive-env.sh 后的效果图，从图中可以看出，编者在该文件中指定了 Hadoop 安装目录：

```
[allen@cmaster0 ~]$ cat apache-hive-1.2.1-bin/conf/hive-env.sh
# Set HADOOP_HOME to point to a specific hadoop install directory
HADOOP_HOME=/home/allen/hadoop-2.7.1
# Hive Configuration Directory can be controlled by:
# export HIVE_CONF_DIR=
# Folder containing extra libraries required for hive compilation/execution can be controlled by:
# export HIVE_AUX_JARS_PATH=
[allen@cmaster0 ~]$
```

图 11-6 指定 Hadoop 安装目录

Step3 在 HDFS 里新建 Hive 存储目录。

在 Hive 运行过程中，其需要使用 “/user/hive/warehouse” 来存储 metadata（元数据），故还要在 HDFS 中新建 Hive 用到的相关存储目录，新建命令及其权限分配如下：

```
[allen@iclient0 hadoop-2.7.1]$ bin/hdfs dfs -mkdir /tmp
[allen@iclient0 hadoop-2.7.1]$ bin/hdfs dfs -mkdir -p /user/hive/warehouse
[allen@iclient0 hadoop-2.7.1]$ bin/hdfs dfs -chmod g+w /tmp
[allen@iclient0 hadoop-2.7.1]$ bin/hdfs dfs -chmod g+w /user/hive/warehouse
```

Step4 启动 Hive 命令行。

在内嵌模式下，由于 Hive 只是 Hadoop 的一个 “智能终端”，所谓的启动 Hive 指的是启动 Hive 运行时环境，用户可使用下述命令进入 Hive 运行时环境：

```
[allen@iclient0 ~]$ cd apache-hive-1.2.1-bin
[allen@iclient0 apache-hive-1.2.1-bin]$ bin/hive
```

由于内嵌模式下的 Hive 只是 Hadoop 的一个 “智能终端”，故只有当进入 Hive 环境或执行 Hive 脚本时，才会启动，故其仅是非驻守进程，用则开，开则启，关则停。

Step5 验证 Hive 是否启动成功。

使用 “bin/hive” 命令进入 Hive 环境后，可使用 “show tables”、“show functions”、“create...” 等命令验证是否部署成功，编者执行验证命令，其效果图如图 11-7 所示。

```
[allen@iclient0 ~]$ cd apache-hive-1.2.1-bin
[allen@iclient0 apache-hive-1.2.1-bin]$ bin/hive
Logging initialized using configuration in jar:file:/home/allen/apache-hive-1.2.1-bin/lib/hive-common-1.2.1.jar!/hive-log4j.properties
hive> show tables;
OK
Time taken: 1.739 seconds
hive> show functions;
OK
!=
%
&
+
-
/
<=
<>
<
=
>
A
abs
acos
add_months
and
array_contains
ascii
```

进入Hive环境

查看有无表

显示所有函数

函数太多，下面未截图

```
[allen@iclient0 ~]$ cd apache-hive-1.2.1-bin
[allen@iclient0 apache-hive-1.2.1-bin]$ bin/hive
Logging initialized using configuration in jar:file:/home/allen/apache-hive-1.2.1-bin/lib/hive-common-1.2.1.jar!/hive-log4j.properties
hive> show tables;
OK
Time taken: 1.769 seconds
hive> show functions;
OK
!=
%
&
+
-
/
<=
<>
<
=
>
A
abs
acos
add_months
and
array_contains
ascii
```

进入hive命令行接口

查看有无hive表

查看内置函数

太多，未截图

图 11-7 进入 Hive 环境

3. 完全模式部署

显然，在内嵌模式下，当只有 Joe 用户使用 iclient0 上的 Hive 时，正常不会发生任

何“意外”，可当 iclient0 上的 Joe、iclient1 上的 Kevin 和 iclient2 上的 Alice 都要同时使用 Hive 且修改的是同一个表时，Metastore 必须支持多个进程。由于内嵌模式下，Hive 中 Metastore 和运行时环境属于同一进程，其无法支持多用户模式，故需要部署完整的 Hive 就必须单独部署“Hive Metastore”和“HiveServer2”。

手工部署完整模式的 Hive 过程较为复杂，编者不再演示，下面只讲述使用 Ambari 部署完全模式的 Hive。

使用 Ambari 部署完全模式的 Hive 可以说是一键操作，难点几乎都在 Ambari 工具本身部署上，以下步骤从无到有，简单介绍了 Ambari 自身部署和使用 Ambari 部署 Hive 的大概步骤：

- Step1 制定部署规划。
- Step2 准备硬件机器和 OS 环境。
- Step3 配置单机 OS 环境和集群环境。
- Step4 部署 Ambari-server。
- Step5 使用 Ambari-server 部署 HDFS、YARN、Hive。

例 1 请使用 Ambari 为 littleCstor 部署完全模式的 Hive。

解 由于大数据平台涉及太多组件，故部署之前最好制定一个完备的部署计划，否则极有可能导致由于各机角色分配混乱而部署失败。

本题以第 3 章为前提，只给出 Hive 部署规划表（表 11-1）和部署效果图（图 11-8），具体部署过程请参见第 3 章。读者须注意，图 11-8 中 iclient0 到 cmaster2 的链接实际上并不存在，也就是 iclient0 并不需要向任何机器汇报心跳包，只有当 iclient0 需要主动连接 Hive 时，其才会主动连接 cmaster2。

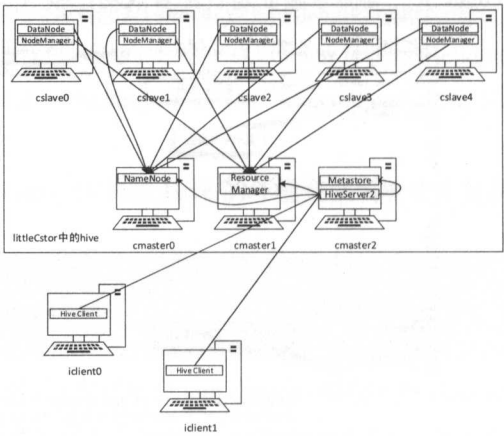


图 11-8 littleCstor 中的完全模式 Hive

表 11-1 littleCstor 上 Hive 部署规划

机 器	角 色	部署服务
cmaster2	服务端	Hive Metastore（元数据服务）
		HiveServer2（运行时服务）
iclient0	客户端	hive shell 命令行接口

11.2 Hive 接口

Hive 接口指的是用户取得 Hive 服务的途径，下面先讲述常见的 Hive 接口，接着直接讲述 Web 接口。

11.2.1 接口汇总

作为大数据处理领域最常用的数据仓库，针对不同的上层应用，Hive 主要提供了如下多种接口：

- Hive Web 接口
- Hive Shell 接口
- Hive API 接口
- Hcatalog 接口
- Pig 接口
- Beeline 接口

Hive Web 接口简称为 HWI (Hive Web Interface)，其是 Hive Shell 接口的一个替代方案，通过该界面，用户可在页面上管理常见的表。

既然 Hive 的核心功能是提供类 SQL 的运行环境，则其 Shell 接口必然是重中之重，通过 Shell 接口，程序员和分析师很容易编写 HiveQL 来实现新建表和查询表操作，11.3 节重点介绍 Hive 的 Shell 接口。

Hive API 面向使用 Java 或 Python 编程的数据分析师，通过该接口，分析师可编写数据库中没有的复杂查询语句。不过，当分析师编写好用户自定义函数后，执行时，一般依旧是在 Hive Shell 接口执行，11.3 节中将会涉及用户自定义函数。

关于通过 Pig、Beeline 和 Hcatalog 获取 Hive 服务，限于篇幅，本章不再讲解。

11.2.2 实战 Hive Web

由于 Hive Web 接口内容较少，此处直接讲解。HWI (Hive Web Interface) 的默认地址为运行 “Hiveserver2IP:9999”，littleCstor 中该地址即为 “cmaster2:9999”，进入该地址

后，可看到类似图 11-9 的界面。

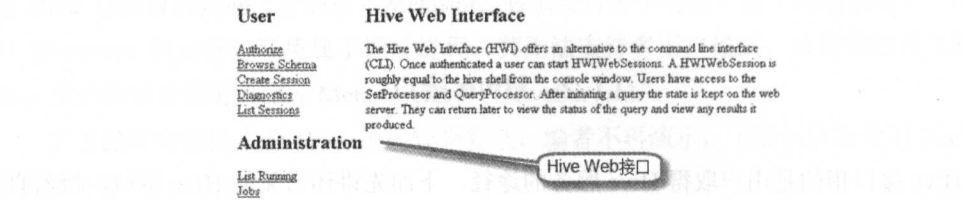


图 11-9 littleCstor 中 Hive Web 接口

限于篇幅，本章不再讲述 HWI 上各类操作，请读者自行练习。

11.3 实战 Hive Shell

对于要学习 Hive 的读者，官方有详细学习文档，本节所有实例均来自该文档，其官方网址为“<https://cwiki.apache.org/confluence/display/Hive/GettingStarted>”。若该地址失效，读者可使用 Google、Bing 或 Baidu，直接搜索“hive”，进入官网后，点击“Getting Started Guide”即可看到本文档。

11.3.1 DDL Operations

常见的 DDL（Data Definition Language）操作包括新建表、显示表、更改表和删除表，下面分别给出实例。

1. 创建 pokes 表

默认情况下，新建表的存储格式均为 Text 类型，字段间默认分隔符为键盘上的 Tab 键，下面为两个建表示例。

```
hive> CREATE TABLE pokes (foo INT, bar STRING);
```

上述命令完成创建有两个字段的 pokes 表，其中，第一个列名为 foo，数据类型为 INT，第二个列名为 bar，数据类型为 STRING。

```
hive> CREATE TABLE invites (foo INT, bar STRING) PARTITIONED BY (ds STRING);
```

上述命令完成创建有两个实体列和一个（虚拟的）分区字段的 invites 表，注意分区字段并不属于 invites 表，当向 invites 导入数据时，ds 字段会用来过滤导入的数据。

2. 显示所有表

```
hive> SHOW TABLES;
```

显然，上述语句和 MySQL 中的 show 很类似，同 MySQL 中操作一样，Hive 也支持正则查询，比如命令只显示以 s 结尾的表：

```
hive> SHOW TABLES '*.s';
```

3. 显示表列

```
hive> DESCRIBE invites;
```

上述命令用来显示表中定义的列项。

4. 更改表

```
hive> ALTER TABLE events RENAME TO 3koobecaf;
```

```
hive> ALTER TABLE pokes ADD COLUMNS (new_col INT);
```

```
hive> ALTER TABLE invites ADD COLUMNS (new_col2 INT COMMENT 'a comment');
```

```
hive> ALTER TABLE invites REPLACE COLUMNS (foo INT, bar STRING, baz INT COMMENT 'baz replaces new_col2');
```

上述第一条语句为将 events 表重命名为 3koobecaf，第二条语句实现向 pokes 中新增一列（列名为 new_col），第三条语句新增列时加了解释。

第四条语句则较为复杂，其会更换原来 invites 表中所有列名，不过，原有的数据并不会发生改变，显然对该语句稍加更改，可实现删除列操作，如：

```
hive> ALTER TABLE invites REPLACE COLUMNS (foo INT COMMENT 'only keep the first column');
```

上述语句实际上删除了 bar 和 baz 两列。

5. 删除表

```
hive> DROP TABLE pokes;
```

11.3.2 DML Operations

DML (Data Manipulation Language) 操作指的是将数据导入 Hive 表。显然，要向 Hive 表中导入数据，首先肯定是有数据，此处直接使用 Hive 提供的测试数据，其位于“/home/allen/apache-hive-1.2.1-bin/examples/files/”文件夹下。

```
hive> LOAD DATA LOCAL INPATH './examples/files/kv1.txt' OVERWRITE INTO TABLE pokes;
```

上述语句实现将 kv1.txt 导入 pokes 表，该命令执行时会覆盖 pokes 中原有数据。其中，'LOCAL'选项指明 kv1.txt 位于本地，若未加此项则默认文件位于 HDFS；'OVERWRITE'

选项表明删除 kv1.txt 表中之前已有数据。

需要注意的是, 该语句执行时 Hive 并不会检测 kv1.txt 中到底有几列, 第一列字段到底是不是 INT 类型, 该语句就是一个申明语句, 仅此而已。此外, 若文件位于 HDFS, 该命令实际上是一个移动操作, 其会将 HDFS 中原路径移至 Hive 数据目录, 比如编者此处 Hive 数据目录前缀为 “/home/allen/hive/warehouse”, 由于表名为 pokes, 故完成 HDFS 中, 完整路径为 “/home/allen/hive/warehouse/pokes/kv1.txt”

```
hive> LOAD DATA LOCAL INPATH './examples/files/kv2.txt' OVERWRITE INTO TABLE invites
PARTITION (ds='2008-08-15');
hive> LOAD DATA LOCAL INPATH './examples/files/kv3.txt' OVERWRITE INTO TABLE invites
PARTITION (ds='2008-08-08');
```

上述语句实现将数据导入 invites 表中两个不同分区。

```
hive> LOAD DATA INPATH '/user/allen/kv2.txt' OVERWRITE INTO TABLE invites PARTITION
(ds='2008-08-15');
```

上述语句将 HDFS 中的 “/user/allen/kv2.txt” 文件导入 invites 表。

11.3.3 SQL Operations

常见 SQL 语句包括 SELECT、GROUP、JOIN 等, 下面逐一以实例讲述。

1. SELECTS and FILTERS

```
hive> SELECT a.foo FROM invites a WHERE a.ds='2008-08-15';
```

查找 invites 表中, foo 列 ds 为 2008-08-15 的所有数据, Hive 会调用 MapReduce 执行该语句并将结果输出到控制台。

```
hive> INSERT OVERWRITE DIRECTORY '/tmp/hdfs_out' SELECT a.* FROM invites a WHERE
a.ds='2008-08-15';
```

和第一条命令功能类似, 不同的是, 其将结果集存入了 HDFS 中的 “/tmp/hdfs_out” 目录。

```
hive> INSERT OVERWRITE LOCAL DIRECTORY '/tmp/local_out' SELECT a.* FROM pokes a;
```

将 pokes 表中所有列的数据都导入本地文件系统的 “/tmp/local_out” 目录下。

```
hive> INSERT OVERWRITE TABLE events SELECT a.* FROM profiles a;
hive> INSERT OVERWRITE TABLE events SELECT a.* FROM profiles a WHERE a.key < 100;
hive> INSERT OVERWRITE LOCAL DIRECTORY '/tmp/reg_3' SELECT a.* FROM events a;
hive> INSERT OVERWRITE DIRECTORY '/tmp/reg_4' select a.invites, a.pokes FROM profiles a;
hive> INSERT OVERWRITE DIRECTORY '/tmp/reg_5' SELECT COUNT(*) FROM invites a
WHERE a.ds='2008-08-15';
hive> INSERT OVERWRITE DIRECTORY '/tmp/reg_5' SELECT a.foo, a.bar FROM invites a;
hive> INSERT OVERWRITE LOCAL DIRECTORY '/tmp/sum' SELECT SUM(a.pc) FROM pokes a;
```

上述命令演示了常见的函数，请读者逐一练习。

2. GROUP BY

```
hive> FROM invites a INSERT OVERWRITE TABLE events SELECT a.bar, count(*) WHERE
a.foo > 0 GROUP BY a.bar;
hive> INSERT OVERWRITE TABLE events SELECT a.bar, count(*) FROM invites a WHERE
a.foo > 0 GROUP BY a.bar;
```

3. JOIN

```
hive> FROM pokes t1 JOIN invites t2 ON (t1.bar = t2.bar) INSERT OVERWRITE TABLE events
SELECT t1.bar, t1.foo, t2.foo;
```

4. MULTITABLE INSERT

```
FROM src
INSERT OVERWRITE TABLE dest1 SELECT src.* WHERE src.key < 100
INSERT OVERWRITE TABLE dest2 SELECT src.key, src.value WHERE src.key >= 100 and
src.key < 200
INSERT OVERWRITE TABLE dest3 PARTITION(ds='2008-04-08', hr='12') SELECT src.key
WHERE src.key >= 200 and src.key < 300
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/dest4.out' SELECT src.value WHERE
src.key >= 300;
```

5. STREAMING

```
hive> FROM invites a INSERT OVERWRITE TABLE events SELECT TRANSFORM(a.foo, a.bar)
AS (oof, rab) USING '/bin/cat' WHERE a.ds > '2008-08-09';
```

11.4 实战 Hive 之复杂语句

本节示例参考“<https://cwiki.apache.org/confluence/display/Hive/GettingStarted>”，该示例是 Hive 进阶的经典例题，若该地址失效，读者可使用 Google、Bing 或 Baidu，直接搜索“hive”，进入官网后，点击“Getting Started Guide”即可看到本文档。

下面的语句主要完成新建表 `u_data` 并向其导入本地某数据，实际操作时，步骤如下。

(1) 创建表

如下语句为实现创建有四个字段的 `u_data` 表：

```
hive> CREATE TABLE u_data (
```

```

>   userid INT,
>   movieid INT,
>   rating INT,
>   unixtime STRING,
>) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE;

```

(2) 导入数据

建好表后，自然而然就想到向表里导入数据，下面的操作实现向 `u_data` 表里导入大量数据。

Step1 准备数据。

如下命令完成下载并解压一机器学习文件：

```

[allen@iclient0 ~]$ wget http://files.grouplens.org/datasets/movielens/ml-100k.zip
[allen@iclient0 ~]$ unzip ml-100k.zip

```

Step2 向 `u_data` 表导入数据。

下述命令实现将解压后的 `u.data` 内容导入 `u_data` 表中。

```

hive> LOAD DATA LOCAL INPATH '/home/allen/ml-100k/u.data' OVERWRITE INTO TABLE
u_data;

```

(3) Select 查询语句

如下语句实现统计表中数据有多少条，该语句执行时，Hive 运行时环境会将该语句翻译成 MapReduce 操作并向 Hadoop 集群提交该操作：

```

hive> SELECT COUNT(*) FROM u_data;

```

上述操作结束，下述命令完成新建 `u_data_new` 表，在建表时调用 Python 脚本并使用 `u_data` 中的数据，执行步骤如下。

(1) 编写 `weekday_mapper.py` 脚本

```

for line in sys.stdin:
    line = line.strip()
    userid, movieid, rating, unixtime = line.split('\t')
    weekday = datetime.datetime.fromtimestamp(float(unixtime)).isoweekday()
    print '\t'.join([userid, movieid, rating, str(weekday)])

```

(2) 新建 `u_data_new` 表

```

hive> CREATE TABLE u_data_new (
>   userid INT,
>   movieid INT,
>   rating INT,
>   weekday INT
>) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

```

(3) 向 `u_data_new` 表中导入 `u_data` 表中的数据

```

hive> add FILE weekday_mapper.py;
hive> INSERT OVERWRITE TABLE u_data_new
> SELECT

```

```
> TRANSFORM (userid, movieid, rating, unixtime)
> USING 'python weekday_mapper.py'
> AS (userid, movieid, rating, weekday)
> FROM u_data;
```

(4) 使用复杂 Select 语句查询 u_data_new 表中数据

```
hive> SELECT weekday, COUNT(*) FROM u_data_new GROUP BY weekday;
```

11.5 实战 Hive 之综合示例

下面的语句完成：①进入 Hive 命令行接口，获取 Hive 函数列表并单独查询 count 函数用法。②在 Hive 里新建 member 表，并将表 11-2 中的数据载入 Hive 里的 member 表中。③查询 member 表中所有记录，查询 member 表中 gender 值为 1 的记录，查询 member 表中 gender 值为 1 且 age 为 22 的记录，统计 member 中男性和女性出现次数。下面按问题顺序依次讲述。

表 11-2 结构化表 member

身份 ID	姓名	性别	年龄	教育	职业	收入
201401	aa	0	21	e0	p3	m
201402	bb	1	22	e1	p2	l
201403	cc	1	23	e2	p1	m

①本问非常简单，参考下面两条命令即可，注意示例中所有操作均在 iclient0 上以 allen 用户执行。

```
[allen@iclient0 ~]# hive #进入 Hive 命令行
hive>show functions; #获取 Hive 所有函数列表
hive>describe function count; #查看 count 函数用法
```

②对于问题②，显然应先为表准备数据，即在 iclient 目录 “/home/allen” 下新建文件 memberData 并写入如下内容，注意记录间为换行符，字段间以 Tab 键分割。

```
201401 aa 0 21 e0 p3 m
201402 bb 1 22 e1 p2 l
201403 cc 1 22 e2 p1 m
```

下面建表时将赋予各个字段合适的含义与类型，由于较为简单，请直接参考下面语句，这里不再赘述。

```
hive>show tables; #查看当前 Hive 仓库中所有表(以确定当前无 member 表)
hive>create table member(id int,name string,gender tinyint,age tinyint,edu string,prof string,income string)row format delimited fields terminated by '\t'; #使用合适字段与类型，新建 member 表
hive>show tables; #再次查看，将显示 member 表
```

```

hive>load data local inpath '/home/allen/memberData' into table member; #将本地文件 m..载入
HDFS
hive>select * from member; #查看表中所有记录
hive>select * from member where gender=1; #查看表中 gender 值为 1 的记录
hive>select * from member where gender=1 AND age=23; #查看表中 gender 值为 1 且 age 为 23 的
记录
hive>select gender,count(*) from member group by gender; #统计男女出现总次数
hive>drop table member; #删除 member 表
hive>quit; #退出 Hive 命令行接口

```

统计表中“男女出现次数”是一个常见的 SQL 操作，其原理和 MapReduce 的 WordCount 相同，显然、Hive 将 Hadoop 抽象成了 SQL 类型的数据仓库。

11.6 实战 Hive API 接口

虽然 HiveQL 内置了 216 个函数，但在某些特殊场景下，可能还是需要自定义函数。Hive 的 UDF 包括三种：UDF (User-Defined Function)、UDAF (User-Defined Aggregate Function) 和 UDTF (User-Defined Table-Generating Function)。普通 UDF 支持一个输入产生一个输出，UDAF 支持多个输入一个输出，UDTF 支持一个输入多个输出。Hive 只支持 Java 编写的 UDF，其他的编程语言只能通过 select transform 转化为流来与 Hive 交互。下面通过两个编程示例说明如何编写 UDF 以及在 Hive 中如何使用 UDF。

11.6.1 UDF 编程示例

UDF 类^[3]必须继承自 org.apache.hadoop.hive.ql.exec.UDF 类，并且实现 evaluate 方法。下面编写一个对查询结果进行大小写转换的 UDF，步骤如下。

Step1 在 eclipse 中新建 Java 工程。

Step2 导入 Hive 的 lib 目录下的 jar 包，本例用到 org.apache.hadoop.io 包，所以也需要将 Hadoop 的 jar 包导入。

Step3 新建包 com.cstore，在包下建立 lower_Or_UpperCase.java，可以有多个 evaluate 方法，依据参数的类型和个数来区分。

Lower_Or_UpperCase.java 代码如下：

```

package com.cstore;
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;
//继承 UDF

```



```

public class lower_Or_UpperCase extends UDF {
    //实现至少一个 evaluate 方法
    public Text evaluate(Text t,String up_or_lower){
        if (t == null){return null;}
        //依据标识参数，转换大小写
        else if (up_or_lower.equals("lowercase")){
            return new Text(t.toString().toLowerCase());}
        else if (up_or_lower.equals("uppercase")){
            return new Text(t.toString().toUpperCase());}
        else
            return null;
    }
}

```

Step4 检查代码无误后打成 jar 包，jar 包名为 uporlower.jar，存放在 /home/allen 目录下。

Step5 进入 Hive 的 shell，用 add jar 命令把 jar 包导入 Hive 的环境变量里，用 create temporary function as 命令基于 jar 包中的类创建临时函数，之后就可以在查询中使用函数了；这一过程执行的命令和部分输出如下。

```

hive> add jar /home/dengpeng/uporlower.jar;
Added /home/dengpeng/uporlower.jar to class path
Added resource: /home/dengpeng/uporlower.jar
hive> create temporary function uporlower as 'com.cstore.lower_Or_UpperCase';
OK
Time taken: 0.0030 seconds
hive> select uporlower(name,'uppercase') from userinfo;
OK
JACK
KAM
.....

```

Step6 最后可以把不再需要的函数销毁。

```
hive> drop temporary function uporlower;
```

11.6.2 UDAF 编程示例

UDAF 类必须继承自 org.apache.hadoop.hive ql.exec.UDAF 类，并且在其内部类中必须实现 org.apache.hadoop.hive ql.exec.UDAFEvaluator 接口，UDAFEvaluator 接口有五个方法，分别为：

- ①init 方法负责对中间结果实现初始化；
- ②iterate 接收传入的参数，并进行内部的轮转，其返回类型为 boolean；

③`terminatePartial` 没有参数, 负责返回 `iterate` 函数轮转后的数据;

④`merge` 接收 `terminatePartial` 的返回结果, 合并接收的中间值, 返回类型为 `boolean`;

⑤`terminate` 返回最终结果。

下面编写一个对查询结果求几何平均值的 UDAF, 代码如下:

```
import org.apache.hadoop.hive ql.exec.UDAF;
import org.apache.hadoop.hive ql.exec.UDAFEvaluator;
import org.apache.hadoop.io.IntWritable;

public class GeometricMean extends UDAF {
    public static class midResult {
        public long numCount;
        public double multSum;
    }

    public static class GMEvaluator implements UDAFEvaluator {
        midResult midr;

        public GMEvaluator() {
            super();
            midr = new midResult();
            init();
        }

        public void init() {
            midr.multSum = 1;
            midr.numCount = 0;
        }

        public boolean iterate(IntWritable a) {
            if (a != null) {
                midr.multSum *= a.get();
                midr.numCount++;
            }
            return true;
        }

        public midResult terminatePartial() {
            return midr.numCount == 0 ? null : midr;
        }

        public boolean merge(midResult b) {
            if (b != null) {
                midr.numCount *= b.numCount;
                midr.multSum += b.multSum;
            }
            return true;
        }

        public Double terminate() {
            return midr.numCount == 0 ? null : Math.pow( midr.multSum, 1.0/midr.numCount);
        }
    }
}
```

UDAF 程序的使用方式与 UDF 是一样的，这里不再详述。

习 题

1. 既然 Hive 任务最终还是翻译成了 MapReduce，为何需要 Hive？
2. 简述 Hive 功能作用及其体系架构。
3. 当 Hive 元数据存储位置在不同位置时，给 Hive 集群带来何种影响？
4. 简述手工部署 Hive、使用 Ambari 部署 Hive 的步骤。
5. 简述 Hive 访问接口。
6. 简述使用 Maven 和不使用 Maven 时，Hive 开发环境搭建部署。
7. 简述常见的 Hive 调优技术。
8. 在大型系统中，Hive 最常使用的场景是什么？
9. 在大型系统中，现有一个 Hive 工作流（如报表），如何定时触发该工作流？
10. 对相同的数据和业务逻辑，试编写使用桶分区的 HiveQL 和不使用桶分区的 HiveQL，试比较两者执行效率。
11. 在大型集群中，如何确保 Hive 集群自身安全性？Hive 数据呢？
12. 在一个大型集群中，对于一个长期运行的 Hive，如何以后台方式运行该工作流？

参考文献

- [1] <http://hive.apache.org/>
- [2] <https://cwiki.apache.org/confluence/display/Hive/GettingStarted>
- [3] <https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF>

CDAT 软件的安装 (CDAT 软件的安装)

1. 安装 CDAT 软件前，请先安装以下软件：
- Windows 95/NT
- Java 1.1 或以上版本

2. 安装 CDAT 软件时，请按照以下步骤操作：

3. 安装 CDAT 软件时，请按照以下步骤操作：

4. 安装 CDAT 软件时，请按照以下步骤操作：

5. 安装 CDAT 软件时，请按照以下步骤操作：

6. 安装 CDAT 软件时，请按照以下步骤操作：

7. 安装 CDAT 软件时，请按照以下步骤操作：

8. 安装 CDAT 软件时，请按照以下步骤操作：

9. 安装 CDAT 软件时，请按照以下步骤操作：

10. 安装 CDAT 软件时，请按照以下步骤操作：

11. 安装 CDAT 软件时，请按照以下步骤操作：

12. 安装 CDAT 软件时，请按照以下步骤操作：

13. 安装 CDAT 软件时，请按照以下步骤操作：

14. 安装 CDAT 软件时，请按照以下步骤操作：

15. 安装 CDAT 软件时，请按照以下步骤操作：

参考文献

[1] Java 1.1 或以上版本

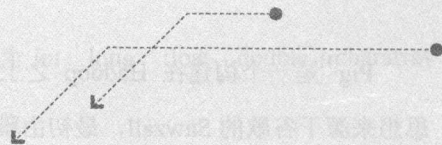
[2] Windows 95/NT

[3] CDAT 软件的安装

和MapReduce一样，它也是为数据仓库环境设计的，如果以它为基础构建数据仓库，那么它就是一个不错的选择。Hadoop是谷歌开发的一个分布式系统，其设计目标是实现大规模数据的存储和计算。Hadoop的架构非常复杂，它由多个组件组成，包括HDFS、MapReduce、Hive、Pig、Tez等。Hadoop的生态系统非常丰富，它支持多种数据类型和查询语言。Hadoop的生态系统还包括Hive、Pig、Tez、Mahout、Hama、HBase、Cassandra、Kafka、Storm、Flink、Spark等。Hadoop的生态系统是一个非常庞大的生态系统，它支持多种数据类型和查询语言。Hadoop的生态系统是一个非常庞大的生态系统，它支持多种数据类型和查询语言。

第12章 其他常见大数据组件

HADOOP
BEING DIGITAL



Hadoop 是谷歌 GFS 和 MapReduce 的开源实现, 其 HDFS 和 YARN 分别为分布式集群提供了最基础的分布式存储和分布式集群资源管理功能, 不过, Hadoop 大数据生态圈还包含一些其他组件, 这些组件一般应用于特定场景 (如数据收集、数据挖掘), 本章即讲述这些实用小组件。

12.1 Pig

Pig^[1]是一个构建在 Hadoop 之上, 用来处理大规模数据集的脚本语言平台。其设计思想来源于谷歌的 Sawzall, 最初由雅虎团队开发, 并于 2008 年 9 月贡献给 Apache。程序员或分析师只需要根据业务逻辑写好数据流脚本, Pig 会将写好的数据流处理脚本翻译成多个 HDFS、Map 和 Reduce 操作。通过这种方式, Pig 为 Hadoop 提供了更高层次的抽象, 将程序员从具体的编程中解放出来。

12.1.1 Pig 简介

1. Pig 基本框架

Pig 相当于一个 Hadoop 的客户端, 它先连接到 Hadoop 集群, 之后才能在集群上进行各种操作。Pig 的基本框架^[2]如图 12-1 所示。

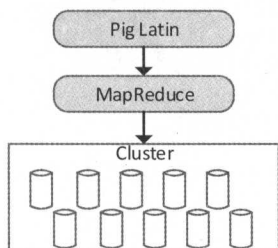


图 12-1 Pig 基本框架

Pig 包括两部分, 一部分是用于描述数据流的语言, 称为 Pig Latin; 另一部分则是用于运行 Pig Latin 程序的执行环境。Pig Latin 程序由一系列的 operation 和 transformation 组成, Pig 内部解释器会将这些变换操作转换成一系列的 HDFS 操作和 MapReduce 作业, 这些操作整体上描述了一个数据流。

当需要处理海量数据时, 先用 Pig Latin 语言编写 Pig Latin 数据处理脚本, 然后在 Pig 中执行 Pig Latin 程序, Pig 会自动将 Pig Latin 脚本翻译成 MapReduce 作业, 上传到集群, 并启动执行。对用户来说, 底层的 MapReduce 工作完全是透明的, 用户只需要了解 SQL-Like 的 Pig Latin 语法, 就可以驱动强大的集群。但 Pig 不适合所有的数据处理任务,

和 MapReduce 一样，它是为数据批处理而设计的。如果只想查询大数据集中的一小部分数据，Pig 的实现不会很好，因为它要扫描整个数据集或绝大部分数据。

2. Pig 语法

Pig Latin 是 Pig 的专用语言，它是类似于 SQL 的面向数据流语言，这套脚本语言提供了对数据进行排序、过滤、求和、分组、关联等各种操作，此外，用户还可以自定义一些函数（User-Defined Functions, UDF），以满足某些特殊的数据处理要求。

1) Pig Latin 数据类型

(1) 基本数据类型

和大部分程序语言类似，Pig 的基本数据类型为 int、long、float、double、chararray 和 bytearray。

(2) 复杂数据类型

字符串或基本类型与字符串的组合，主要包含下述四种。

Filed: 存放一个原子类型数据，如一个字符串或一个数字等，例如'lucy'。

Tuple: Filed 的序列，其中每个 File 可以是任何一种基本类型，例如 ('lucy', '1234')。

Bag: Tuple 集合。每个 Tuple 可以包含不同数目不同类型的 Filed，例如：

('lucy', '1234')
('jack' ('ipod', 'apple'))

Map: 一组键值对的组合，一个关系中的键值对必须是唯一的，例如：

[name#Mike,phone#18362100000]

2) Pig Latin 运算符

Pig Latin 提供了算术、比较、关系等运算符，这些运算符的含义和用法与其他语言 (C, Java) 相差不大。其中算术运算符主要包括加 (+)，减 (-)，乘 (*)，除 (/)，取余 (%) 和三目运算符 (?:)，比较运算符主要包括等于 (==)，不等 (!=)。

3) Pig Latin 函数

Pig Latin 是由一系列函数（命令）构成的数据处理流，这些函数或是内置或是用户自定义，表 12-1 是最常用的几个命令。

表 12-1 Pig 常用命令

操作名称	功能
LOAD	载入待处理数据
FOREACH	逐行处理 Tuple
FILTER	过滤不满足条件的 Tuple
DUMP	将结果打印到屏幕
STORE	将结果保存到文件

3. Pig 部署

Pig 支持手工和工具两种部署方式，当以手工方式部署 Pig 时，由于 Pig 只相当于 Hadoop 的一个客户端，用户所写的 Pig Latin 经翻译器翻译后再提交集群执行，故只要在客户机上部署 Pig 即可；当使用 Ambari 部署 Pig 时，同样只需要在客户机上部署 Pig 客户端即可。本节不讲述手工部署方式，只给出 Ambari 部署 Pig 总体步骤。

使用 Ambari 部署 Pig 可以说是一键操作，难点几乎都在 Ambari 工具本身部署上，以下步骤从无到有，简单介绍了 Ambari 自身部署和使用 Ambari 部署 Pig 的大概步骤：

- Step1 制定部署规划。
- Step2 准备硬件机器和 OS 环境。
- Step3 配置单机 OS 环境和集群环境。
- Step4 部署 Ambari-server。
- Step5 使用 Ambari-server 部署 HDFS、YARN、Pig。

例 1 请使用 Ambari 为 littleCstor 部署 Pig。

解 由于大数据平台涉及太多组件，故部署之前最好制定一个完备的部署计划，否则极有可能导致由于各机角色分配混乱而部署失败。

表 12-2 littleCstor 上 Pig 部署规划

机器	角色	部署服务
iclient0	终端	Pig 解析器

本题以第 3 章为前提，只给出 Pig 部署规划表（表 12-2）和部署效果图（图 12-2），具体部署过程请参见第 3 章。读者须注意，图 12-2 中 Pig 到 NameNode 和 ResourceManager 的连接（图中 Ø 表示）实际上并不存在，也就是 iclient0 并不需要向任何机器汇报心跳包，只有当 iclient0 需要使用 HDFS 或 YARN 时，它才会主动连接它们。

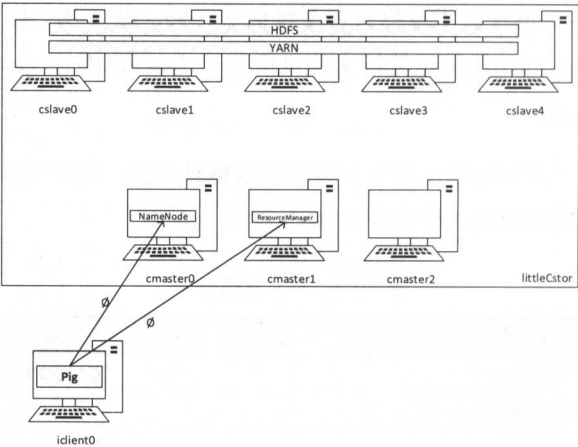


图 12-2 Pig 基本框架

12.1.2 实战 Pig

Pig 提供了类 Shell 方式的访问接口，用户在 Linux Shell 下输入 pig，然后回车即可进入 Pig 命令行接口（即 grunt）。

下述命令完成①进入 Pig 命令行，查看并练习常用命令。②使用 Pig Latin 实现 WordCount，该程序处理 HDFS 上/user/allen/input 中的文件。

```
[allen@iclient0 ~]# bin/pig           #进入 allen 用户的 Pig 命令行
grunt> help;                          #查看 Pig 操作
grunt> A = load 'input';              #载入待处理文件夹 input
grunt> B = foreach A generate flatten(TOKENIZE((chararray)$0)) as word;    #划分单词
grunt> C = group B by word;           #指定按单词聚合，即同一个单词到一起
grunt> D = foreach C generate COUNT(B),group; #同一个单词出现次数相加
grunt> store D into 'out/wc-19';       #将处理好的文件存入 HDFS 下
/user/allen/out/wc-19
grunt> dump D into ;                  #将处理结果 D 打印到屏幕
```

执行时，用户可以将结果存入 HDFS，也可以将结果打印到屏幕。注意，只有最后两条语句才会触发 MapReduce 程序，这种“懒”策略有利于提高集群利用率。

12.2 Oozie

Oozie^[3]起源于雅虎，主要用于管理与组织 Hadoop 工作流。Oozie 的工作流必须是一个有向无环图，实际上 Oozie 就相当于 Hadoop 的一个客户端，当用户需要执行多个关联的 MapReduce (MR) 任务时，只需要将 MR 执行顺序写入 workflow.xml，然后使用 Oozie 提交本次任务，Oozie 会托管此任务流。

12.2.1 Oozie 简介

现实业务中处理数据时不可能只包含一个 MR 操作，一般都是多个 MR，并且中间还可能包含多个 Java 或 HDFS，甚至是 Shell 操作，利用 Oozie 可以完成这些任务。实际上 Oozie 不是仅用来配置多个 MR 工作流的，它可以是各种程序夹杂在一起的工作流，比如执行一个 MR1 后，接着执行一个 Java 脚本，再执行一个 Shell 脚本，接着是 Hive 脚本，然后又是 Pig 脚本，最后又执行了一个 MR2，使用 Oozie 可以轻松完成这种多样

的工作流，使用 Oozie 时，若前一个任务执行失败，后一个任务将不会被调度。

Oozie 的主要功能包括：组织各种工作流（包括 Pig、Hive 等），以规定方式执行工作流（包括定时任务、定数任务、数据促发任务等），托管工作流（包括命令行接口，任务失败时的通知机制，如邮件通知等）。

由于需要存储工作流信息，为提供高可靠性，确保任务配置不丢失，Oozie 内部使用数据库来存储工作流相关信息，用户可以使用 Oozie 内嵌的 Derby 数据库，也可以使用 MySQL、PostgreSQL、Oracle 等数据库，为降低复杂性，本节部署时使用内嵌的 Derby 数据库。

1. Oozie 部署

Oozie 相当于 Hadoop 的一个客户端，因此集群中只有一台机器部署 Oozie server 端即可，由于可以有任意多个客户端连接 Oozie，故每个客户端上都须部署 Oozie client，本节选择在 cmaster0 上部署 Oozie server，在 iclient0 上部署 Oozie client。

Oozie 支持手工和工具两种部署方式，不过手工部署 Oozie 时须配置太多参数，故本节只讲述使用 Ambari 部署 Oozie。

使用 Ambari 部署 Oozie 可以说是一键操作，难点几乎都在 Ambari 工具本身部署上，以下步骤从无到有，简单介绍了 Ambari 自身部署和使用 Ambari 部署 Oozie 的大概步骤：

- Step1 制定部署规划。
- Step2 准备硬件机器和 OS 环境。
- Step3 配置单机 OS 环境和集群环境。
- Step4 部署 Ambari-server。
- Step5 使用 Ambari-server 部署 HDFS、YARN、Oozie。

例 2 请使用 Ambari 为 littleCstor 部署 Oozie。

解 由于大数据平台涉及太多组件，故部署之前最好制定一个完备的部署计划，否则极有可能导致由于各机角色分配混乱而部署失败。

本题以第 3 章为前提，只给出 Oozie 部署规划表（表 12-3）和部署效果图（图 12-3），具体部署过程请参见第 3 章。读者须注意，图 12-3 中 Oozie Client 到 Oozie Server 的连接（图中 Ø 表示）实际上并不存在，也就是 Oozie Client 并不需要向任何机器汇报心跳包，只有当 iclient0 需要使用 Oozie 服务时，它才会主动连接 Oozie Server。

表 12-3 littleCstor 上 Oozie 部署规划

机 器	角 色	部署服务
cmaster0	服务端	oozie Server
iclient0	终端	oozie Client

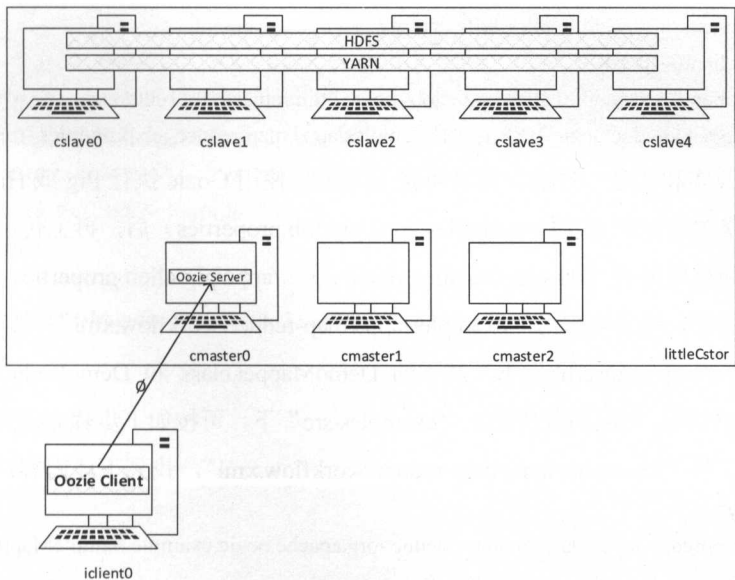


图 12-3 littleCstor 中的 Oozie

12.2.2 实战 Oozie

Oozie 最常用的是命令行接口^[4]，它的 Web 接口只可以看到 Oozie 托管的任务，不可以配置作业。

按要求完成问题：①进入 Oozie 客户端，查看常用命令。②运行 Oozie MR 示例程序。③运行 Oozie Pig、Hive 等示例。④编写 workflow.xml，完成一次 WordCount。⑤编写 workflow.xml，完成两次 WordCount，且第一个 WC 的输出为第二个 WC 的输入。

对于问题①，在 iclient0 上执行下述命令即可。

```
[allen@iclient0 ~]# oozie help #查看所有 Oozie 命令
```

对于问题②，首先解压 Oozie 示例 jar 包，接着修改示例配置中的地址信息，最后上传至集群执行即可，读者按下述流程执行即可。

```
[allen@iclient0 ~]# cd oozie-VSERISON
[allen@iclient0 ~]# tar -zxvf oozie-examples.tar.gz
```

编辑 examples/apps/map-reduce/job.properties,将如下两行：

```
nameNode=hdfs://localhost:8020
jobTracker=localhost:8021
```

替换成集群现在配置的地址与端口：

```
nameNode=hdfs://cmaster0:8020
jobTracker=cmaster01:8032
```

接着将 examples 上传至 HDFS，使用 oozie 命令执行即可：

```
[allen@iclient0 ~]# sudo -u allen hdfs dfs -put examples examples
[allen@iclient0 ~]# cd
[allen@iclient0 ~]# sudo -u allen oozie job -oozie http://cmaster0:11000/oozie
-config /usr/share/doc/oozie-VERSION/examples/apps/map-reduce/job.properties -run
```

问题③其实和②是一样的,读者可按上述过程使用 Oozie 执行 Pig 或 Hive 等的示例脚本。切记修改相应配置(如 examples/apps/pig/job.properties)后,再上传至集群,执行时也要定位到相应路径(如 sudo -u allen oozie /apps/pig/allen.properties -run)。

对于问题④,读者可参考“examples/apps/map-reduce/workflow.xml”,其对应 jar 包在“examples/apps/map-reduce/lib”下,其下的 DemoMapper.class 和 DemoReducer.class 就是 WordCount 的代码,对应的源代码在“examples/src”下,可按如下步骤完成此问题。

①编辑文件“examples/apps/map-reduce/workflow.xml”,找到下述内容:

```
<property>
<name>mapred.mapper.class</name><value>org.apache.oozie.example.SampleMapper</value>
</property>
<property>
<name>mapred.reducer.class</name><value>org.apache.oozie.example.SampleReducer</value>
</property>
```

②将其替换成:

```
<property>
<name>mapred.mapper.class</name><value>org.apache.oozie.example.DemoMapper</value>
</property>
<property>
<name>mapred.reducer.class</name><value>org.apache.oozie.example.DemoReducer</value>
</property>
<property><name>mapred.output.key.class</name><value>org.apache.hadoop.io.Text</value></prop
erty>
<property>
  <name>mapred.output.value.class</name><value>org.apache.hadoop.io.IntWritable</value>
</property>
```

③接着将原来 HDFS 里 examples 文件删除,按问题②的解答,上传执行即可,这里只给出删除原 examples 的命令,上传和执行命令和问题②解答一样。

```
[allen@iclient0 ~]# sudo -u allen hdfs dfs -rm -r -f examples #删除 HDFS 原 examples 文件
```

问题⑤是业务逻辑中最常遇到的情形,比如你的数据处理流是:“M1”→“R1”→“Java1”→“Pig1”→“Hive1”→“M2”→“R2”→“Java2”,单独写出各类或脚本后,写出此逻辑对应的 workflow.xml 即可。限于篇幅,下面只给出 workflow.xml 框架,请读者自行解决问题④。

```
<workflow-app xmlns="uri:oozie:workflow:0.2" name="map-reduce-wf">
  <start to="mr-node"/>
  <action name="mr-node">
```

```

    <map-reduce>第一个 wordcount 配置</map-reduce>
    <ok to="mr-wc2"/><error to="fail"/>
  </action>
  <action name="mr-wc2">
    <map-reduce>第二个 wordcount 配置</map-reduce>
    <ok to="end"/><error to="fail"/>
  </action>
  <kill name="fail">
    <message>Map/Reduce failed error message[${wf.errorMessage(wf.lastErrorNode())}]
  </message>
  </kill>
  <end name="end"/>
</workflow-app>

```

12.3 Flume

Flume^[5]是一个分布式高性能、高可靠的数据传输工具，它可用简单的方式将不同数据源的数据导入某个或多个数据中心，典型应用是将众多生产机器日志数据实时导入 HDFS。除了简单的数据传输功能外，Flume 更像一个智能的路由器，内部提供了强大的分用、复用、断网续存功能。

12.3.1 Flume 简介

1. Flume 逻辑结构

Flume 核心思想是数据流，即数据从哪来到哪去，中间需不需要经过谁。比如将生产机 WebA 和 WebB 的日志数据实时导入 HDFS，须在 WebA、WebB 和集群中部署 Flume，WebA 与 WebB 上的 Flume 负责读取并实时发送日志，集群中的 Flume 则负责接收数据并将数据写入 HDFS（图 12-4）。

用户可以将 Flume 看成是两台机器之间通过网络互相传送数据，甚至用户自己可以使用 netty 写一个类似程序（实际上 Flume 内部也是封装 netty 实现的），不同之处在于 Flume 定制了大量的数据源（如 Thrift、Shell）与数据汇（如 Thrift、HDFS、Hbase），用户只要简单配置即可使用。此外，通过使用“管道”，Flume 能够确保不会丢失一条数据，

提供了数据高可靠性，即使在断网的情况下，Flume 也会将数据先存入“管道”，待网络恢复后重新发送，图 12-5 是 Flume 逻辑图。

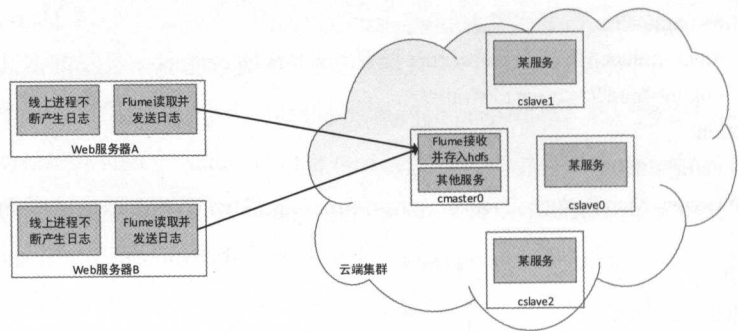


图 12-4 Flume 典型应用

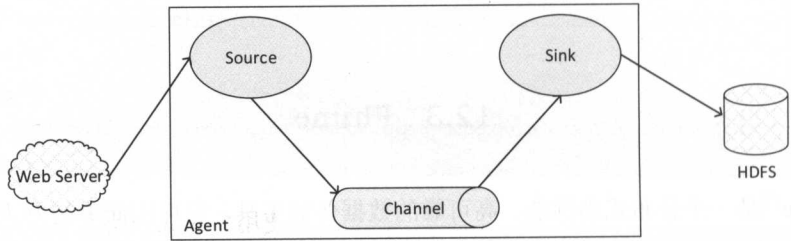


图 12-5 Flume 逻辑图

2. Flume 组成

Flume 包含 Source、Channel 和 Sink 三个组成部分，且这三部分是相互关联的，配置时须在配置文件里申明这三部分，并指定所属关系，下面简单介绍这三个组件。

1) Source

它负责读取原始数据，目前 Flume 支持 Avro Source, Thrift Source, Exec Source（即 Shell），NetCat Source, Syslog Sources, HTTP Source 等大量类型，甚至用户可以自定义 Source，使用时在配置文件里声明即可。

2) Channel

它负责将从 Source 端传来的数据存入 Channel，目前 Flume 包含三种类型的 Channel，即 Memory, JDBC 和 File，当传输数据量特别大时，用户应当考虑使用 File Channel，当然用户也可以自定义 Channel，同 Source 一样，使用时，在配置文件中指定即可。此外，Flume 的分用、复用和过滤功能即在于此，通过定义并控制多个相互无关的 Channel，可以实现数据发往不同地点而并不干涉。

3) Sink

它负责从 Channel 中取出并发送数据, Flume 当前支持 HDFS、Logger、Avro、Thrift、IRC、File 和 Hbase 等大量类型 Sink, 其实这些 Sink 内部都是使用 netty 来发送数据的, 只是发送的协议不同而已。

Flume 将 Source、Channel 和 Sink 构成的统一体称为 Agent, 启动时须以 Agent 为单位启动 Flume。

3. Flume 部署

集群中只有一台机器部署 Flume 就可以接收数据了, 此外下面的实例中还要有一台机器做为数据源, 负责向 Hadoop 集群发送数据, 故须在 cmaster0 与 iclient0 上部署 Flume。

12.3.2 实战 Flume

Flume 提供了命令行接口和程序接口, 但 Flume 使用方式比较特别, 无论是命令行还是程序接口, 都必须使用 Flume 配置文档, 这也是 Flume 架构思想之一——配置型工具。

下面给出一实际场景并在此场景下给出相关 Flume 操作: ①进入 Flume 命令行, 查看常用命令。②要求发送端 iclient0 使用 telnet 向 cmaster0 发送数据, 而接收端 cmaster0 开启 44444 端口接收数据, 并将收到的数据显示于命令行。③要求发送端 iclient0 将本地文件 “/home/allen/source.txt” 发往接收端 cmaster0, 而接收端 cmaster0 将这些数据存入 HDFS。

对于问题①, 直接在 iclient0 上执行如下命令即可:

```
[allen@iclient0 ~]# flume-ng
```

#查看 Flume 常用命令

对于问题②, 首先需要在 cmaster0 上按要求配置并开启 Flume (作为接收进程被动接收数据), 接着在 iclient0 上使用 telnet 向 cmaster0 发送数据, 具体过程参见如下几步。

在 cmaster0 上以 allen 权限, 新建文件 “/home/allen/flume/conf/flume.conf”, 并填入如下内容:

```
# 命令此处 agent 名为 a1, 并命名此 a1 的 sources 为 r1, channels 为 c1, sinks 为 k1
a1.sources = r1
a1.channels = c1
a1.sinks = k1
# 定义 sources 相关属性: 即此 sources 在 cmaster0 上开启 44444 端口接收以 netcat 协议发来的数据
a1.sources.r1.type = netcat
```

```

a1.sources.r1.bind = cmaster0
a1.sources.r1.port = 44444
# 定义 channels 及其相关属性, 此处指定此次服务使用 memory 暂存数据
a1.channels.c1.type = memory
a1.channels.c1.capacity = 1000
a1.channels.c1.transactionCapacity = 100
# 定义此 sink 为 logger 类型 sink: 即指定 sink 直接将收到的数据输出到控制台
a1.sinks.k1.type = logger
# 将 sources 关联到 channels, channels 关联到 sinks 上
a1.sources.r1.channels = c1
a1.sinks.k1.channel = c1

```

接着在 cmaster0 上使用此配置以前台方式开启 Flume 服务:

```
[allen@cmaster0 ~]# flume-ng agent -c /etc/flume-ng/ -f /home/allen/flume-ng/conf/flume.conf -n
```

a1

此时, 接收端 cmaster0 已经配置好并开启了, 接下来需要开启发送端, 在 iclient0 上执行:

```
[allen@iclient0 ~]# telnet cmaster0 44444
```

此时向此命令行里随意输入数据并回车, telnet 会将这些数据发往 cmaster0, 再次回到 cmaster0 上执行命令的那个终端, 会发现刚才在 iclient0 里输入的数据发送到了 cmaster0 的终端里。如果想退出 iclient0 终端里的 telnet, 按 Ctrl+]组合键 (即同时按住 Ctrl 键和]键), 回到 telnet 后输入 “quit” 命令回车即可, 至于退出 cmaster0 上的 Flume, 直接按 Ctrl+C 组合键。

问题③的回答步骤较多。

首先, 在 cmaster0 上新建文件 “/home/allen/flume-ng/conf/flume.conf.hdfs”, 并填入如下内容:

```

# 命令此处 agent 名为 a1, 并命名此 a1 的 sources 为 r1, channels 为 c1, sinks 为 k1
a1.sources = r1
a1.sinks = k1
a1.channels = c1
# 定义 sources 类型及其相关属性
# 即此 sources 为 avro 类型, 且其在 cmaster0 上开启 4141 端口接收 avro 协议发来的数据
a1.sources.r1.type = avro
a1.sources.r1.bind = cmaster0
a1.sources.r1.port = 4141
# 定义 channels 类型及其相关属性, 此处指定此次服务使用 memory 暂存数据
a1.channels.c1.type = memory
# 定义此 sink 为 HDFS 类型的 sink, 且此 sink 将接收的数据以文本方式存入 HDFS 指定目录
a1.sinks.k1.type = hdfs
a1.sinks.k1.hdfs.path = /user/allen/flume/cstorArchive
a1.sinks.k1.hdfs.fileType = DataStream

```

```
# 将 sources 关联到 channels, channels 关联到 sinks 上
```

```
a1.sources.r1.channels = c1
```

```
a1.sinks.k1.channel = c1
```

接着, 在 iclient0 上新建文件 “/home/allen/businessLog”, 并填入如下内容:

```
cccccccccccccccccccccc
```

```
ssssssssssssssssssssss
```

```
tttttttttttttttttttttt
```

```
oooooooooooooooooooo
```

```
rrrrrrrrrrrrrrrrrrrrrr
```

iclient0 上还要新建文件 “/home/allen/flume-ng/conf/flume.conf.exce”, 并填入如下内容:

```
# 命令此处 agent 名为 a1, 并命名此 a1 的 sources 为 r1, channels 为 c1, sinks 为 k1
```

```
a1.sources = r1
```

```
a1.channels = c1
```

```
a1.sinks = k1
```

```
# 定义 sources 类型及其相关属性, 此 sources 为 exce 类型
```

```
# 其使用 Linux cat 命令读取文件/allen/businessLog, 接着将读取到的内容写入 channel
```

```
a1.sources.r1.type = exec
```

```
a1.sources.r1.command = cat /allen/businessLog
```

```
# 定义 channels 及其相关属性, 此处指定此次服务使用 memory 暂存数据
```

```
a1.channels.c1.type = memory
```

定义此 sink 为 avro 类型 sink, 即其用 avro 协议将 channel 里的数据发往 cmaster0 的 4141 端口

```
a1.sinks.k1.type = avro
```

```
a1.sinks.k1.hostname = cmaster0
```

```
a1.sinks.k1.port = 4141
```

```
# 将 sources 关联到 channels, channels 关联到 sinks 上
```

```
a1.sources.r1.channels = c1
```

```
a1.sinks.k1.channel = c1
```

至此, 发送端 iclient0 和接收端的 Flume 都已配置完成。现在需要做的是在 HDFS 里新建目录, 并分别开启接收端 Flume 服务和发送端 Flume 服务, 步骤如下。

在 cmaster0 上开启 Flume, 其中 “flume-ng ... a1” 命令表示使用 flume.conf.hdfs 配置启动 Flume, 参数 a1 即是配置文件里第一行定义的那个 a1。

```
[allen@cmaster0 ~]# hdfs dfs -mkdir flume #HDFS 里新建目录/user/allen/flume
```

```
[allen@cmaster0 ~]# flume-ng agent -c /etc/flume-ng/ -f /home/allen/flume-ng/conf/flume.conf.hdfs -n a1
```

最后, 在 iclient0 上开启发送进程, 与上一条命令类似, 这里的 a1, 即 flume.conf.exce 定义的 a1:

```
[allen@iclient0 ~]# flume-ng agent -c /etc/flume-ng/ -f /etc/flume-ng/conf/flume.conf.exce -n a1
```

此时, 用户在 iclient0 端口里打开 “cmaster0:50070”, 依次进入目录 “/user/allen/flume/cstorArchive”, 将会查看到从 iclient0 上传送过来的文件。

12.4 Mahout

Mahout^[6]是基于 Hadoop 平台的机器学习工具,它提供了大量机器学习算法的 MR 实现,此外,它还提供了大量针对数据预处理的工具类,通过数据预处理工具类与机器学习算法的结合,能够很方便地实现从模型构建到性能测试等一系列步骤。

12.4.1 Mahout 简介

目前 Mahout 主要包含分类、聚类和协同过滤三种类型算法,需要注意的是 Mahout 算法处理的数据类型必须是矩阵类型的二进制数据,若数据为文本类型,用户须通过 Mahout 提供的数据转换工具完成转换,接着提交给相关算法,用户可以把 Mahout 看成一个 Hadoop 客户端,只是这个客户端包含了大量的机器学习包。

作为 Hadoop 的一个客户端, Mahout 只要在集群中或集群外某台客户机上部署即可,实验中选择在 iclient0 上部署 Mahout。请读者自行下载并解压 Mahout,下面命令为编者在 iclient0 上解压 Mahout:

```
[allen@iclient0 ~]# tar -zxvf Desktop/apache-mahout-distribution-0.11.1.tar.gz
```

12.4.2 实战 Mahout

Mahout 提供了程序和命令行接口,通过参考 Mahout 已有的大量机器学习算法,程序员也可实现将某算法并行化。

下面以 allen 用户运行 Mahout 示例程序 naivebayes,实现下载数据,建立学习器,训练学习器,最后使用测试数据针对此学习器进行性能测试。

对于上述问题,首先须下载训练数据集和测试数据,接着运行训练 MR 和测试 MR,但是, Mahout 里的算法要求输入格式为 Value 和向量格式的二进制数据,故中间还须加一些步骤,将数据转换成要求格式的数据,下面的脚本 naivebayes.sh 可以完成这些动作。

```
#!/bin/sh
#新建本地目录,新建 HDFS 目录
mkdir -p /tmp/mahout/20news-bydate /tmp/mahout/20news-all && hdfs dfs -mkdir mahout
#下载训练和测试数据集
```



```

curl http://people.csail.mit.edu/jrennie/20Newsgroups/20news-bydate.tar.gz \
-o /tmp/mahout/20news-bydate.tar.gz
#将数据集解压、合并，并上传至 HDFS
cd /tmp/mahout/20news-bydate && tar xzf /tmp/mahout/20news-bydate.tar.gz && cd
cp -R /tmp/mahout/20news-bydate/*/* /tmp/mahout/20news-all
hdfs dfs -put /tmp/mahout/20news-all mahout/20news-all
#使用工具类 seqdirectory 将文本数据转换成二进制数据
/home/allen/apache-mahout-distribution-0.11.1/bin/mahout seqdirectory -i mahout/20news-all -o
mahout/20news-seq -ow
#使用工具类 seq2sparse 将二进制数据转换成算法能处理的矩阵类型二进制数据
/home/allen/apache-mahout-distribution-0.11.1/bin/mahout seq2sparse -i mahout/20news-seq -o
mahout/20news-vectors -lnorm -nv -wt tfidf
#将总数据随机分成两部分，第一部分约占总数据 80%，用来训练模型
#剩下的约 20%作为测试数据，用来测试模型
/home/allen/apache-mahout-distribution-0.11.1/bin/mahout split -i mahout/20news-vectors/tfidf-vectors--
trainingOutput mahout/20news-train-vectors \
--testOutput mahout/20news-test-vectors \
--randomSelectionPct 40 --overwrite --sequenceFiles -xm sequential
#训练 Naive Bayes 模型
/home/allen/apache-mahout-distribution-0.11.1/bin/mahout trainnb -i mahout/20news-train-vectors
-el -o mahout/model -li mahout/labelindex -ow
#使用训练数据集对模型进行自我测试（可能会产生过拟合）
/home/allen/apache-mahout-distribution-0.11.1/bin/mahout testnb -i mahout/20news-train-vectors
-m mahout/model -l mahout/labelindex -ow -o mahout/20news-testing
#使用测试数据对模型进行测试
/home/allen/apache-mahout-distribution-0.11.1/bin/mahout testnb -i mahout/20news-test-vectors -m
mahout/model -l mahout/labelindex -ow -o mahout/20news-testing

```

限于篇幅，脚本较简洁，执行时，切记须在 `iclient0` 上，以 `allen` 用户身份执行，且只能执行一次。再次执行时，先将所有数据全部删除，执行方式如下：

```

[allen@iclient0 ~]# cp naivebayes.sh /home/allen
[allen@iclient0 ~]# chown allen.allen naivebayes.sh
[allen@iclient0 ~]# chmod +x naivebayes.sh
[allen@iclient0 ~]# sh naivebayes.sh

```

脚本执行时，用户可以打开 Web 界面“`cmaster0:8088`”，查看正在执行的 Mahout 任务；还可以通过 Web 界面“`cmaster0:50070`”，定位到“`/user/allen/mahout/`”查看目录变化。

习 题

1. 既然已经有了 Hive, 为什么还需要 Pig? 既然 Hive 和 Pig 最终都是调用了 MapReduce, 为什么还出现了 Hive 和 Pig?
2. 编写 Pig 脚本, 读取 Text、Sequence 类文件, 是否可以使用 Pig 读取嵌套列结构?
3. Pig 内置的三角函数, 是否是使用 MapReduce 执行的?
4. 如何使用 Pig 操作 HBase 或 Hive 里的数据?
5. 现有一工作流, 其处理过程为: “Flum→HiveQL→HBase→Python1→Pig1→Spark-App→MR-App→Java1”, 试使用 Oozie, 将这些工作流组织成一个 Oozie 流。
6. 对于上述 Oozie 流, 如何配置 Oozie 按时触发器, 按数据触发器又如何配置?
7. 假设有 100 台生产机器, 现在要使用 Flume 将这些生产机的日志导入两个不同的数据中心, 如何设计数据流?
8. Flume 的 Channel 有各种类型, 其中有一个为内存型, 试问当 Flume 采用内存型 Channel 时, 如何保证内存占用量?
9. Flume 中, 为什么会有分用、复用技术, 这些技术有哪些应用场景?
10. Flume 采用了何机制来确保数据安全性?
11. 针对 Mahout 里聚类、分类和推荐三类算法, 从每类中任意选一个算法在 Mahout 中执行。
12. 试分别在 MapReduce、Spark 和 H2O 上执行 Mahout 里的 KMeans 算法, 执行结束后, 试比较三者执行效率。

参考文献

- [1] <http://pig.apache.org/>
- [2] <http://pig.apache.org/docs/r0.15.0/>
- [3] <http://oozie.apache.org/>
- [4] <http://oozie.apache.org/docs/4.2.0/index.html>
- [5] <http://flume.apache.org/FlumeUserGuide.html>
- [6] <http://mahout.apache.org/>

1. 制定部署规划

在开始部署规划前，先明确部署目标，将之前章节的 Hadoop 部署方案与本章的部署方案进行对比，明确部署目标。部署规划主要包含以下三个方面：部署 Hadoop 的版本、部署 Hadoop 的节点数、部署 Hadoop 的节点类型。

① 部署 Hadoop 的版本

无论是上一个章节的部署方案，还是本章的部署方案，部署 Hadoop 的版本都是 Hadoop 2.0。部署 Hadoop 的版本是 Hadoop 2.0，部署 Hadoop 的版本是 Hadoop 2.0。

2. 部署环境

部署环境

部署环境是指部署 Hadoop 2.0 所需的硬件和软件环境。部署环境包括以下几个方面：部署 Hadoop 2.0 的硬件环境、部署 Hadoop 2.0 的软件环境、部署 Hadoop 2.0 的网络环境、部署 Hadoop 2.0 的安全环境、部署 Hadoop 2.0 的运维环境。

附录 A

手工部署 Hadoop2.0

HADOOP
BEING DIGITAL

部署 Hadoop 2.0 的硬件环境是指部署 Hadoop 2.0 所需的硬件设备。部署 Hadoop 2.0 的硬件环境包括以下几个方面：部署 Hadoop 2.0 的服务器、部署 Hadoop 2.0 的存储设备、部署 Hadoop 2.0 的网络设备、部署 Hadoop 2.0 的安全设备、部署 Hadoop 2.0 的运维设备。

部署 Hadoop 2.0 的软件环境是指部署 Hadoop 2.0 所需的软件环境。部署 Hadoop 2.0 的软件环境包括以下几个方面：部署 Hadoop 2.0 的操作系统、部署 Hadoop 2.0 的数据库、部署 Hadoop 2.0 的中间件、部署 Hadoop 2.0 的运维工具、部署 Hadoop 2.0 的安全工具、部署 Hadoop 2.0 的运维平台。

部署 Hadoop 2.0 的网络环境是指部署 Hadoop 2.0 所需的网络环境。部署 Hadoop 2.0 的网络环境包括以下几个方面：部署 Hadoop 2.0 的网络设备、部署 Hadoop 2.0 的网络配置、部署 Hadoop 2.0 的网络安全、部署 Hadoop 2.0 的网络运维、部署 Hadoop 2.0 的网络平台。

部署 Hadoop 2.0 的安全环境是指部署 Hadoop 2.0 所需的安全环境。部署 Hadoop 2.0 的安全环境包括以下几个方面：部署 Hadoop 2.0 的安全设备、部署 Hadoop 2.0 的安全配置、部署 Hadoop 2.0 的安全策略、部署 Hadoop 2.0 的安全运维、部署 Hadoop 2.0 的安全平台。

部署 Hadoop 2.0 的运维环境是指部署 Hadoop 2.0 所需的运维环境。部署 Hadoop 2.0 的运维环境包括以下几个方面：部署 Hadoop 2.0 的运维工具、部署 Hadoop 2.0 的运维平台、部署 Hadoop 2.0 的运维人员、部署 Hadoop 2.0 的运维流程、部署 Hadoop 2.0 的运维文档、部署 Hadoop 2.0 的运维培训。

部署 Hadoop 2.0 的部署方案是指部署 Hadoop 2.0 所需的部署方案。部署 Hadoop 2.0 的部署方案包括以下几个方面：部署 Hadoop 2.0 的部署目标、部署 Hadoop 2.0 的部署规划、部署 Hadoop 2.0 的部署实施、部署 Hadoop 2.0 的部署验收、部署 Hadoop 2.0 的部署运维、部署 Hadoop 2.0 的部署培训。

部署 Hadoop 2.0 的部署方案是指部署 Hadoop 2.0 所需的部署方案。部署 Hadoop 2.0 的部署方案包括以下几个方面：部署 Hadoop 2.0 的部署目标、部署 Hadoop 2.0 的部署规划、部署 Hadoop 2.0 的部署实施、部署 Hadoop 2.0 的部署验收、部署 Hadoop 2.0 的部署运维、部署 Hadoop 2.0 的部署培训。

部署 Hadoop 是学习与使用 Hadoop 的必由之路,也是拦路虎,令初学者望而却步。本节将深入浅出地讲解手工部署 Hadoop,和第三章遥相呼应,期望能把读者引入 Hadoop 的精彩世界。

一、部署综述

1. 部署方式

Hadoop 主要有两种部署方式,传统解压包方式和 Linux 标准方式。早期的 Hadoop 都是采用直接解压 `hadoop-x.gz` 包方式部署的,近两年来由于 Cloudera、Hortonworks 等公司对 Hadoop 及其相关组件的包装、整合, Hadoop 部署方式正向标准 Linux 部署方式靠拢。相对来说,标准 Linux 部署方式简单易用,而传统部署方式则烦琐易错,但标准部署方式隐藏了太多细节,相反传统解压包方式有助于读者深入理解 Hadoop,编者建议在采用标准方式部署前,先学习传统部署方式。

此外,无论是解压包方式还是标准方式, Hadoop 部署都有单机模式、伪分布模式和分布式模式,考虑到实战意义,加之为避免混淆,编者只介绍分布式模式。下一小节将以传统解压包方式部署 Hadoop,关于工具部署(标准方式)请读者参阅第 3 章。

2. 部署步骤

无论是解压包方式还是标准方式,部署 Hadoop 时都大概分为如下几个步骤:

- ①制定部署规划;
- ②准备机器;
- ③准备机器软件环境;
- ④下载 Hadoop;
- ⑤解压 Hadoop;
- ⑥配置 Hadoop;
- ⑦启动 Hadoop;
- ⑧测试 Hadoop。

这里称步骤②、③为部署前工作,步骤⑤、⑥、⑦为 Hadoop 部署,最后的步骤⑧为 Hadoop 测试,当然了,其实最重要的还是第①步部署规划,它为 Hadoop 部署指明了方向,根据上述划分, Hadoop 部署步骤又可简述如下:

①制定部署规划;

②部署前工作;

③部署 Hadoop;

④测试 Hadoop。

无论是下一小节的传统部署方式, 还是第 3 章使用的 Ambari 部署, 都会按照这个步骤部署, 请读者务必从整体上把握部署步骤。

3. 准备环境

准备环境讲解的是准备机器和准备机器软件环境, 也就是部署前工作, 本质上说, Hadoop 部署和这一步无关, 但大部分用户或是没有 Linux 环境, 或是刚安装 Linux, 直接使用刚安装的 Linux 来部署完全模式的 Hadoop 是不可能实现的, 用户必须做些诸如修改机器名、添加域名映射等工作 (当然, 若有 DNS 服务器, 那可以不添加域名映射) 后才可部署。

1) 硬件环境

由于分布式计算需要用到很多机器, 部署时用户须提供多台机器, 至于提供几台, 须根据步骤 1 “部署规划” 确定, 如下一节传统方式部署的 “部署规划” 指明使用 3 台机器。

实际上, 完全模式部署 Hadoop 时, 最低需要两台机器 (一个主节点, 一个从节点) 即可实现完全分布模式部署, 而使用多台机器部署 (一个主节点, 多个从节点), 会使这种完全分布模式体现得 “更加充分”, 这二者并无本质区别。读者可以根据自身情况, 做出符合当前实际的 “部署规划”, 其他部署步骤都相同。此外, 硬件方面, 每台机器最低要求有 1GB 内存, 20GB 硬盘空间。

需要特别说明的是, 分布式模式部署中需要使用的机器并非一定是物理实体机器, 实际上, 用户可以提供两台或多台实体机器, 也可以提供两台或多台虚拟机器, 即用户可以使用虚拟化技术, 将一台机器虚拟成两台或多台机器, 并且虚拟后的机器和实体机器使用上无任何区别, 用户可认为此虚拟机就是实体机器。

例 1 机器 A 的配置为 4GB 内存、双核、硬盘 100GB; 系统为 64 位 Windows 7, 现要求使用 VMware 将此机器虚拟成三台 CentOS 机器 cmaster, cslave0, cslave1。

解 用户须下载并安装 VMware, 接着使用 VMware 安装 CentOS。正如在 Windows 7 上安装其他软件一样, 用户根据实际情况, 大体步骤如下。

步骤 1, 下载 VMware Workstation: 谷歌搜索并下载 VMware Workstation。

步骤 2, 安装 VMware Workstation: 在 Windows 7 下正常安装 VMware Workstation

软件。

步骤 3, 下载 CentOS: 到 CentOS 官网下载 64 位的 CentOS, 请尽量下载最新版。

步骤 4, 新建 CentOS 虚拟机: 打开 VMware Workstation→File (文件)→New Virtual Machine Wizard (新建虚拟向导)→Typical (推荐)→Installer disc image file (iso) (例如选中下载的 CentOS-6.7-x.iso 文件)→填写用户名与密码, 用户名建议使用 allen, 密码亦建议使用 allen→填入机器名 cmaster→直至 Finish。

步骤 5, 重复步骤 4, 填入机器名 cslave0, 接着安装直至结束; 再次重复步骤四, 填入机器名 cslave1, 接着安装直至结束。

上述步骤四使用 VMware 新装了 cmaster, 步骤 5 其实跟步骤 4 一样, 只是机器名改成了 cslave0 和 cslave1, 至此, Windows 7 下已新装了三台 CentOS 机器。

需要注意的是, 此处的 cmaster 只是 VMware 面板对此机器的称号, 并不是此机器真实机器名, 实际上新安装 CentOS 的机器名统一为 “localhost.localdomain”, 也就是这三台机器真实机器名都是 “localhost.localdomain”, 而不是 cmaster 或 cslave, 它只是 VMware 面板对这些机器的称号。

此外, 采用虚拟化技术时, 最稀缺的是内存资源, 根据编者经验, 如果你的 Windows 7 机器内存仅为 2GB 时, 其下 VMware 可启动 1 台 CentOS; 4GB 时, VMware 可同时启动 3 台 CentOS; 6GB 时, VMware 可同时启动 5 台 CentOS。此外, 32 位 Windows 7 仅支持 2GB 内存, 如果你的内存大于 2GB, 须使用 64 位 Windows 7。

2) 软件环境

Hadoop 支持 Windows 和 Linux, 但在 Windows 上仅测试过此软件可运行, 并未用于生产实践, 而大量的实践证明, 在 Linux 环境下使用 Hadoop 则更加稳定高效。本节使用 Linux 较成熟的发行版 CentOS 部署 Hadoop, 须注意的是新装系统 (CentOS) 的机器不可以直接部署 Hadoop, 须做些设置后才可部署, 这些设置主要为: 修改机器名, 添加域名映射, 关闭防火墙, 安装 JDK。

例 2 现有一台刚装好 CentOS 系统的机器, 且装机时用户名为 allen, 要求将此机器名修改为 cmaster, 添加域名映射, 关闭防火墙, 并安装 JDK。

解 修改机器名、添加域名映射、关闭防火墙和安装 JDK 这四个操作是 Hadoop 部署前必须做的事情, 请务必做完这四个操作后再部署 Hadoop, 读者可参考如下命令完成这四个操作。

①修改机器名。

```
[allen@localhost ~]$ su - root
```

#切换到 root 用户修改机器名

```
[root@localhost ~]# vim /etc/sysconfig/network
```

#编辑存储机器名文件

将 “HOSTNAME=localhost.localdomain” 中的 “localhost.localdomain” 替换成需要

使用的机器名，按题目要求，此处应为 `cmaster`，即此行内容为：

```
HOSTNAME=cmaster #指定本机名为 cmaster
```

注意重启机器后更名操作才会生效，用户须通过此命令修改集群中所有机器的机器名，重启后，本机将有自己唯一的机器名 `cmaster` 了。

②添加域名映射。

首先使用如下命令查看本机 IP 地址，这里以 `cmaster` 机器为例。

```
[root@cmaster ~]# ifconfig #查看 cmaster 机器 IP 地址
```

假如看到此机器的 IP 地址为 “192.168.1.100”，机器名为 `cmaster`，则域名映射应为：

```
192.168.1.100 cmaster
```

接着编辑域名映射文件 “`/etc/hosts`”，将上述内容加入此文件。

```
[root@cmaster ~]# vim /etc/hosts #编辑域名映射文件
```

③关闭防火墙。

CentOS 的防火墙 `iptables` 默认情况下会阻止机器间通信，编者建议系统管理员开启 Hadoop 使用的端口，也可以暂时关闭或永久关闭 `iptables`（不建议），本节为简单起见，永久关闭防火墙，其关闭命令如下（执行命令后务必重启机器才可生效）：

```
[root@cmaster ~]# chkconfig --level 35 iptables off #永久关闭 iptables，重启后生效
```

④安装 JDK。

Hadoop 部署前须安装 JDK，而且 Hadoop 只能使用 Oracle 的 1.6 及以上版本的 JDK，不能使用 `openjdk`。用户须首先下载 `jdk-x.rpm` 包，如 `jdk-7u40-linux-x64.rpm`。打开刚才已经安装的 CentOS 机器，将 `jdk-7u40-linux-x64.rpm` 复制至虚拟机下某位置，Termianl 下执行（此方式安装的 `jdk` 无须配置 `java_home`）如下命令：

```
[root@cmaster ~]# java #查看 java 是否安装
[root@cmaster ~]# rpm -ivh /home/allen/jdk-7u40-linux-x64.rpm #以 root 权限，rpm 方式安装 JDK
[root@cmaster ~]# java #验证 java 是否安装成功
```

例 3 现有三台机器，且都刚安装好 CentOS 系统，安装系统时用户名皆为 `allen`，要求将此三台机器的名字分别修改为 `cmaster`、`cslave0` 和 `cslave1`，接着添加域名映射，关闭防火墙，并安装 JDK。

解 除了添加域名映射外，其他三项按例 2 根据实际情况，在每台机器上执行即可。此处的域名映射需要在三台机器上都添加，首先登录到每台机器上，查看这三台机器对应的 IP 地址。

```
[root@cmaster ~]# ifconfig #查看 cmaster 机器 IP 地址
[root@cslave0 ~]# ifconfig #查看 cslave0 机器 IP 地址
[root@cslave1 ~]# ifconfig #查看 cslave1 机器 IP 地址
```

假定这三台机器对应的 IP 地址为：

```
192.168.1.100 cmaster
192.168.1.101 cslave0
```



```
192.168.1.102 cslave1
```

接着分别编辑每台机器的“/etc/hosts”文件，将上述内容添加进此文件即可，注意三台机器都要添加。

```
[root@cmaster ~]# vim /etc/hosts           #编辑 cmaster 的域名映射文件
[root@cslave0 ~]# vim /etc/hosts           #编辑 cslave0 的域名映射文件
[root@cslave1 ~]# vim /etc/hosts           #编辑 cslave1 的域名映射文件
```

添加域名映射后，用户就可以在 cmaster 上直接 ping 另外两台机器的机器名了，如：

```
[root@cmaster ~]# ping cslave1             #在 cmaster 上 ping 机器 cslave1
```

4. 关于 Hadoop 依赖软件

Hadoop 部署前提仅是完成修改机器名、添加域名映射、关闭防火墙和安装 JDK 这四个操作，其他都不需要。下面的 ssh 可能是部分读者关心，但实际上却完全不相关的操作或设置。

许多人都认为部署 Hadoop 需要建立集群 ssh 无密钥认证，事实上并不是这样的，ssh 只是给/sbin/start-yarn.sh 等几个 start-x.sh 与 stop-x.sh 脚本使用，Hadoop 本身是一堆 Java 代码，而 Java 代码本身并不依赖 ssh，也完全不应该依赖 ssh（第三方软件），只是运维时为了方便启动或关闭整个集群，才须打通 ssh，本节部署时将不会涉及任何 ssh 操作，也无须打通 ssh。

此外，本节使用的 Hadoop 版本为稳定版 hadoop-2.7.1.tar.gz，读者可以在 Apache 官网下载该版本的 Hadoop。CentOS 版本为 64 位 CentOS-6.7，读者可以到 CentOS 官网下载。JDK 大版本为 jdk-7u40-linux-x64.rpm，读者可以到 Oracle 官网下载。

二、部署步骤

相对于标准 Linux 方式，解压包方式部署 Hadoop 有利于用户更深入理解 Hadoop 体系架构，建议先采用解压包方式部署 Hadoop，熟悉后可采用标准 Linux 方式部署 Hadoop。以下将采用例题的方式，实现在三台机器上部署 Hadoop。

例 4 现有三台机器，且它们都刚装好 64 位 CentOS-6.7，安装系统时用户名为 allen，请按要求完成：①修改三台机器名为 cmaster，cslave0 和 cslave1，并添加域名映射、关闭防火墙和安装 JDK。②以 cmaster 作为主节点，cslave0 和 cslave1 作为从节点，部署 Hadoop。

解 按上一小节讲解的部署步骤，读者可按如下步骤完成部署。

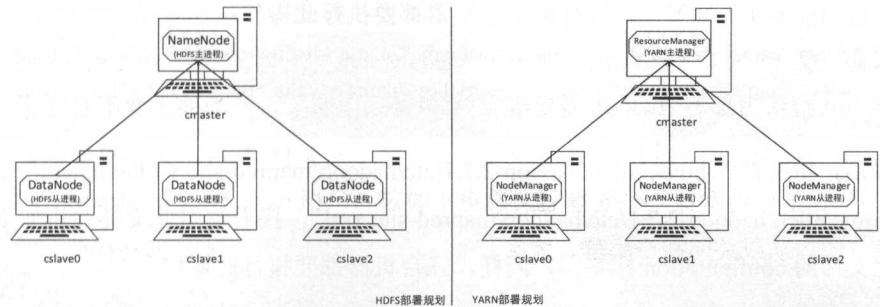
①制定部署规划。

按题目要求，此 Hadoop 集群需三台机器（cmaster，cslave0 和 cslave1），其中 cmaster 作为主节点，cslave0 和 cslave1 作为从节点（表附-1）。

表附-1 hadoop2.0 部署规划表

HDFS	主节点	cmaster	namenode
	从节点	cslave0,cslave1	datanode
YARN	主节点	cmaster	ResourceManager
	从节点	cslave0,cslave1	NodeManager

图附-1 为按此规划，部署后的效果图，图中，编者加入了 cslave2，如果读者机器充足，也可以再添加一台 cslave2，仅有 cslave0 和 cslave1 也是可以的。



图附-1 Hadoop2.0 部署规划图

②准备机器。

请读者准备三台机器，它们可以是实体机也可以是虚拟机，若使用虚拟机，读者可按例 1 新建三台虚拟机。

③准备机器软件环境。

三台机器都要完成：修改机器名、添加域名映射、关闭防火墙和安装 JDK。这几步请参考例 2 与例 3 完成。

④下载 Hadoop。

谷歌搜索“hadoop download”并下载，以 allen 用户身份，将 Hadoop 分别复制到三台机器上。

⑤解压 Hadoop。

分别以 allen 用户登录三台机器，每台都执行如下命令解压 Hadoop 文件：

```
[allen@cmaster ~]# tar -zxvf /home/allen/hadoop-2.7.1.tar.gz #cmaster 上 allen 用户解压 Hadoop
[allen@cslave0 ~]# tar -zxvf /home/allen/hadoop-2.7.1.tar.gz #cslave0 上 allen 用户解压 Hadoop
[allen@cslave1 ~]# tar -zxvf /home/allen/hadoop-2.7.1.tar.gz #cslave1 上 allen 用户解压 Hadoop
```

⑥配置 Hadoop（三台机器都要配置，且配置相同）。

首先, 编辑文件 “/home/allen/hadoop-2.7.1/etc/hadoop/hadoop-env.sh”, 找到如下一行:

```
export JAVA_HOME=${JAVA_HOME}
```

将这行内容修改为:

```
export JAVA_HOME=/usr/java/jdk1.7.0_40
```

这里的 “/usr/java/jdk1.7.0_40” 就是 JDK 安装位置, 如果不同, 读者须根据实际情况更改, 需要注意的是, 三台机器都要执行此操作。

接着, 编辑文件 “/home/allen/hadoop-2.7.1/etc/hadoop/core-site.xml”, 并将如下内容嵌入此文件里 configuration 标签间, 和上一个操作相同, 三台机器都要执行此操作:

```
<property><name>hadoop.tmp.dir</name><value>/home/allen/cloudData</value></property>
<property><name>fs.defaultFS</name><value>hdfs://cmaster:8020</value></property>
```

编辑文件 “/home/allen/hadoop-2.7.1/etc/hadoop/yarn-site.xml”, 并将如下内容嵌入此文件里 configuration 标签间, 同样, 三台机器都要执行此操作:

```
<property><name>yarn.resourcemanager.hostname</name><value>cmaster</value></property>
<property><name>yarn.nodemanager.aux-services</name><value>mapreduce_shuffle</value></pr
operty>
```

最后, 将文件 “/home/allen/hadoop-2.7.1/etc/hadoop/mapred-site.xml.template” 重命名为 “/home/allen/hadoop-2.7.1/etc/hadoop/mapred-site.xml”, 接着编辑此文件并将如下内容嵌入此文件的 configuration 标签间, 同样, 三台机器都要执行此操作:

```
<property><name>mapreduce.framework.name</name><value>yarn</value></property>
```

⑦启动 Hadoop。

首先, 在主节点 cmaster 上格式化主节点命名空间:

```
[allen@cmaster ~]# hadoop-2.7.1/bin/hdfs namenode -format #格式化主节点命名空间
```

接着, 在主节点 cmaster 上启动存储主服务 namenode 和资源管理主服务 resourcemanager。

```
[allen@cmaster ~]# hadoop-2.7.1/sbin/hadoop-daemon.sh start namenode #cmaster 启动存储主服务
[allen@cmaster ~]# hadoop-2.7.1/sbin/yarn-daemon.sh start resourcemanager #启动资源管理主服务
```

最后, 在从节点上启动存储从服务 datanode 和资源管理从服务 nodemanager, 注意, cslave0 和 cslave1 这两台机器上都要执行, 对应命令如下:

```
[allen@cslave0 ~]# hadoop-2.7.1/sbin/hadoop-daemon.sh start datanode #cslave0 启动存储从服务
[allen@cslave0 ~]# hadoop-2.7.1/sbin/yarn-daemon.sh start nodemanager #cslave0 启动资源管理从服务
[allen@cslave1 ~]# hadoop-2.7.1/sbin/hadoop-daemon.sh start datanode #cslave1 启动存储从服务
[allen@cslave1 ~]# hadoop-2.7.1/sbin/yarn-daemon.sh start nodemanager #cslave1 启动资源管理从服务
```

⑧测试 Hadoop。

读者可以分别在三台机器上执行如下命令, 查看 Hadoop 服务是否已启动。

```
$ /usr/java/jdk1.7.0_40/bin/jps #jps 查看 java 进程
$ ps -ef | grep java #ps 查看 java 进程
```

你会在 cmaster 上看到类似的如下信息:

3056 ResourceManager	#资源管理主服务
2347 NameNode	#存储主服务

而在 cslave0 和 cslave1 上看到类似的如下信息:

4021 DataNode	#存储从服务
2761 NodeManager	#资源管理从服务

此外, 还可以任选一台机器, 如 cmaster, 打开 CentOS 默认浏览器 Firefox, 地址栏输入 “cmaster:50070”, 即可在 Web 界面看到 HDFS 相关信息; 同理, 地址栏输入 “cmaster:8088”, 即可在 Web 界面看到 YARN 相关信息。

需要注意的是, 进程显示出来, Web 界面也能看到, 但这并不代表集群部署成功, 一个典型的例子是这些都显示出来, 但做 MapReduce 程序时却出错, 因此我们还要进一步用程序验证集群, 这将在下个例题中讲解。

例 5 使用刚创建的集群, 完成下列要求: ①使用 Hadoop 命令在集群中新建文件夹 “/in”。②将 cmaster 上, 文件夹 “/home/allen/hadoop-2.7.1/etc/hadoop/” 里的所有文件上传至集群的文件夹 “/in” 下。③使用示例程序 WordCount, 统计 “/in” 下每个单词出现次数, 并将结果存入 “/out” 目录。

解 在 cmaster 上, 以 allen 用户, 按如下步骤执行即可:

```
[allen@cmaster ~]# cd hadoop-2.7.1
[allen@cmaster ~]# bin/hdfs dfs -mkdir /in #集群里新建 in 目录
[allen@cmaster ~]# bin/hdfs dfs -put /home/allen/hadoop-2.7.1/etc/hadoop/* /in #将本地文件上传至 HDFS
[allen@cmaster ~]# bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.2.0.jar wordcount /in /out/wc-01 #使用示例程序 WordCount 计算数据
```

此时浏览器迅速打开 “cmaster:8088”, 将会看到 Web 界面上显示正在运行的 Word Count 信息。打开 “cmaster:50070” 并点击链接 “Browse the filesystem” 将会看到刚才的输入数据 “/in” 和输出结果数据 “/out/wc-01/part-r-00000”。当然也可以用 Shell 查看输入输出, 对应的 Shell 命令分别为:

```
[allen@cmaster ~]# bin/hdfs dfs -cat /in/* #使用命令查看 HDFS 中的文件
[allen@cmaster ~]# bin/hdfs dfs -cat /out/wc-01/*
```

至此, Hadoop 部署才算真正完毕。细心的读者会发现, 其实我们根本未涉及任何打通 ssh 操作, 读者应当明白打通 ssh 只是为了 sbin/start-x.sh 相关脚本使用, 并不是 Hadoop 需要的, Hadoop 依赖的只是 Oracle 版 JDK。

通过上述单机部署和集群部署, 可以看出, Hadoop 本身部署起来很简单, 其大量工作其实都是前期的 Linux 环境配置, Hadoop 安装只是解压、修改配置文件、格式化、启动和验证, 关于 Linux 命令问题, 请参考 Linux 专业书籍。



刘鹏

教授、博导、学科带头人，清华大学博士。现任中国云计算专家咨询委员会秘书长、中国信息协会大数据分会副会长、工业和信息化部云计算研究中心专家。

主持完成科研项目25项，发表论文80余篇，出版专业书籍15本。获部级科技进步二等奖4项、三等奖4项。主编了国内第一本云计算教材《云计算》和第一本云计算编程书籍《实战Hadoop》。创办了知名的中国云计算（chincloud.cn）和中国大数据（thebigdata.cn）网站。

曾率队夺得2002 PennySort国际计算机排序比赛冠军，两次夺得全国高校科技比赛最高奖，并三次夺得清华大学科技比赛最高奖。

荣获“全军十大学习成才标兵”（排名第一）、南京“十大杰出青年”、江苏省“333高层次人才培养工程”中青年科学技术带头人、清华大学“学术新秀”等称号。

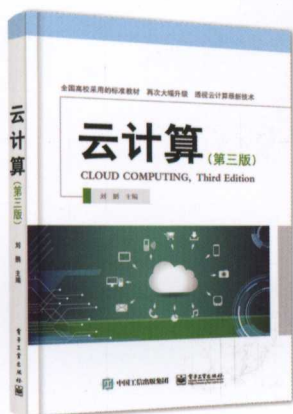


“刘鹏看未来”微信公众号

本书介绍

本书以实战方式，从标准Hadoop集群搭建讲起，全面系统地介绍了Hadoop及其生态圈组件在大数据收集、存储、分析、应用等方面的解决方案，主要包括Ambari、HDFS、YARN、MapReduce、Spark、Storm、ZooKeeper、Hive等内容。

书中实例丰富，操作性强，便于上手。



本书是国内销量最大、被众多高校采用的教材《云计算》的最新升级版，是中国云计算专家咨询委员会秘书长刘鹏教授团队的心血之作。

“让学习变得轻松”是本书的初衷，通过本书可掌握云计算的概念和原理，学习主要的云计算平台和技术，还可了解云计算核心算法和发展趋势。



策划编辑：董亚峰 (dyf@phei.com.cn)
责任编辑：董亚峰 (微信号：sundyf)
封面设计：朝天世纪

ISBN 978-7-121-28564-6



定价：79.00元